PROTECTING PERSISTENT DATA WITH RUN-TIME INVARIANT CHECKING

by

Daniel Fryer

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

Abstract

Protecting Persistent Data with Run-time Invariant Checking

Daniel Fryer
Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto
2019

This thesis describes how we use run-time invariant checking to protect data from corruption originating at high levels in the storage stack. In concert with existing techniques for protection against corruption in the *lower* levels of the storage stack, we make guarantees of data structure consistency in file systems and other storage applications.

We address the *when*, *what*, and *how* of run-time invariant checking. The fundamental requirement is that the application is *crash consistent*, ensuring there is a suitable point in logical time when it is safe to check the invariants. The consistency point provides us a point in time when we *can* check invariants, and an opportunity to check the invariants before the corrupt writes take effect.

Every file system, database, or other storage application has its own data structures, and hence its own consistency invariants. We emphasize checking incrementally because of the scale of data in persistent storage. Our systems check updates for consistency using differential invariants, which are derived from global invariants. Differential invariants compare the old and new states to determine precisely what needs to be checked so the global invariants are not violated by the update. They rely on the consistency of the previous state, so our checking relies on storage systems using other mechanisms to ensure checked data is preserved. In addition to consistency invariants, we also check the atomicity and durability invariants of file system crash consistency mechanisms.

Our evaluation shows the overheads of incremental checking are reasonable. For the Ext3 and Btrfs file systems, the performance overhead of runtime consistency checking is largely masked by device latency. For persistent memory, we see a more substantial overhead, from 10%-50% in our Redis benchmarks. These numbers are also significantly affected by the choice of persistent memory framework adopted, and this is an opportunity for future improvement as more efficient frameworks are developed.

Acknowledgements

This thesis would not have been possible without the support of many friends, family and coworkers. First I would like to thank my supervisors, Ashvin Goel and Angela Demke Brown. Your initial impetus, thoughtful criticism, and excellent writing and editing were even more necessary to my success than your signatures at the end. Ashvin, I have enjoyed (and benefited from) your energy and optimism, as well as your inexhaustible fountain of ideas and inevitable reordering of my paragraphs. Angela, I have likewise learned a lot from your insightful questions, careful writing, and high standards for benchmarking. Thanks also to my committee members Bianca Schroeder and Michael Stumm for their supervision and feedback.

I would also like to thank my coauthors, especially Jack (Kuei) Sun and Mike (Dai) Qin, fellow graduate students who worked on the Recon project with me. I've enjoyed conversations with both of you as well as benefited from your hard work. I have also been the beneficiary of several undergraduate research projects. Many thanks to: Rahat Mahmood, who did early work on Recon and the metadata corruption framework; Shaun Benjamin, who did much of the work on the code for interpreting the Ext3 journal format; Tim (TingHao) Cheng, who helped extract the consistency rules from Btrfs; Kah Wai Lee, who implemented atomicity and durability checks for Ext3; Ekansh Sharma, who enabled Recon to interpose on iSCSI; Andrei Soltan, who implemented the metadata bitmap for Ext3; and Artem Radzhikovskyy, who adapted Recon to Ext4/jbd2. May you all have the best of fortune in your future endeavours! In the same vein, a big thanks to Haryadi Gunawi, who provided us with an SQL version of e2fsck consistency checks which were an incredible help in writing our consistency invariants.

Graduate school was more fun than I had hoped, which might explain how long I spent there. I owe much of that to my friends in SysLab. In no particular order, Michael Mior, Peter Feiner, George Amvrosiadis, Phillipa Gill, Ioan Stefanovici, Andy Hwang, Svitlana Tumanova, Alexey Tumanov, Nosayba El-Sayed, Bogdan Simion, Olya Irzak, Nilton Bila, Sahil Suneja, Suprio Ray, Stathis Maneas: you all made my time at the University of Toronto an enjoyable one and I am grateful for your friendship. There are many more names of friends I've made in the Systems & Networking Lab, the Department, and across Toronto, whose support, advice, and friendship are invaluable.

I owe my appetite for learning to my parents, Bruce & Grace. Your wisdom, encouragement, and patience while I explain file system consistency continues to be appreciated. To my siblings Margaret and Michael, I owe you thanks for not getting me kidnapped while traveling in distant lands, and not getting yourselves kidnapped either; a grad student can't exactly afford ransom.

For the last decade I have had a constant friend, who has become my editor, critic, general sounding board, and wife. Thank you so much, Sarah, for putting up with the distance and time this project has taken!

Lastly, I thank God for the time and opportunity I have had to pursue what interests me, and for His mercy and grace throughout.

Contents

Chapter 1

Introduction

Bugs that corrupt data cause frustration and anger for users and exact a high cost on developers. This is especially problematic if the corruption becomes *persistent*, causing for example a damaged file system, or a corrupted database. Once persistent state has been corrupted, programs may repeatedly crash, hang, be left open to exploit, or propagate the corruption to other nodes in a distributed system [67]. Recovering from such corruption is challenging, requiring sophisticated recovery programs [34], replaying execution non-deterministically [65], or resorting to using backups or snapshots of an older version of the data. Any of these methods may result in the loss of data that was previously reported to the user as durable.

The pathway from an application to physical storage media is complex. It is referred to as the *storage stack*, because of the many layers of components between a user and their data. Figure 1.1 depicts a simplified view of the layers in a typical storage stack. Real world examples may include additional layers, for example, the network connection between a file system and a remote storage device, or a file system running on a virtual disk, nested within another file system. Malfunction in any of these components can result in the corruption or loss of data.

Today, most techniques that enhance the reliability of storage systems focus on detecting and recovering from failures in the *lower levels* of the storage stack. For example, replicating the same data across multiple disks offers protection against whole disk failure or individual media errors. File systems that keep checksums of file contents make it possible to detect corruption originating at any level below the file system. However, none of the methods designed for tolerating errors in the lower levels of the storage stack address corruption originating in higher levels of the storage stack.

The distinction is that bugs in the higher levels *produce* incorrect output, whereas lower level bugs *fail to preserve* the output of the higher levels. Determining that some content generated at a higher level is incorrect requires knowledge of the semantics of that content. We distinguish between the upper and lower layers of the storage stack by designating the upper layers as *storage applications* (Figure 1.1.) Examples of storage applications include file systems, relational databases, and NoSQL databases (e.g., key/value stores). Other programs use these storage applications to store and retrieve data reliably and efficiently.

The complexity of storage applications means that they are vulnerable to bugs that corrupt data. Even though popular file systems are one of the most extensively tested parts of the storage stack, they regularly manifest bugs, as shown by researchers [64,85] and painful real-world experiences [59]. A comprehensive study recently showed that 40% of file system bugs have severe consequences which lead to in-memory or on-disk data corruption [52].

The most widely used remedy for corruption is infrequent, offline checking, such as the tool *fsck* for Linux file systems. Unfortunately, this is time consuming and scales very poorly with data size (on the order of hours or

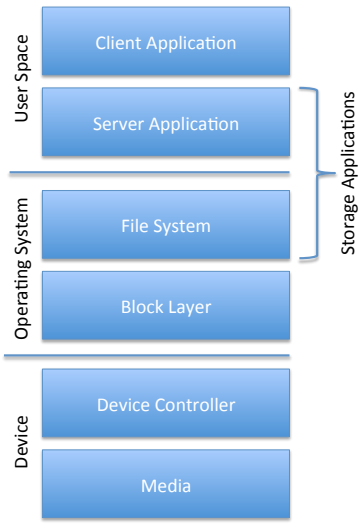


Figure 1.1: The Storage Stack

even days for large volumes); additionally, it is too late to stop corrupt data from overwriting good data.

Offline file system consistency checking is slow enough to be relegated to a disaster recovery operation for large systems, and is performed infrequently as a maintenance operation on desktop systems. Likewise, applications such as Microsoft Exchange and various DBMSs include check & repair tools, but there may be days or weeks between the time that corruption occurs and the time of detection.

Failure to detect corruption shortly after it occurs results in further damage. The corrupt state may be propagated to additional storage or backups, or cause incorrect operation. When detection comes long after the point when the corruption occurs it is also difficult to attribute corruption to a particular cause. It may have been the result of a software bug, a hardware bug, a configuration error or malice. Because of the difficulty in distinguishing between hardware- and software- caused corruption, some developers will not even accept bug reports showing a corrupt heap unless the user is already using ECC memory. Similarly, Microsoft will only investigate a stack trace showing a crash in Windows after they have multiple reports of an identical crash. The earlier we detect a bug, the easier it is to perform root cause analysis, or to recover before reporting incorrect results to a user.

Once corruption is discovered, it requires complex recovery procedures. Real-world solutions fall into two categories, both of which are unsatisfactory. One approach is to use disaster recovery methods, such as a backup or a snapshot, but these can cause significant downtime and loss of recent data. Another option is to use an offline repair tool to restore the consistency of the corrupt state. While this is used in practice for file systems and databases (e.g., e2fsck for the Ext3 file system), it still may cause data loss. Consistency repair tools are known to be difficult to write, and are themselves prone to bugs which can worsen the damage [33].

The rapidly increasing scale of persistent data, the complexity of the storage applications, and the risks of data corruption due to bugs which continue to evade detection call for new ways to prevent corruption originating higher in the storage stack.

One way to prevent bugs is formal specification and verification. Recently, two verified storage systems have been proposed [12, 72]. So long as the behaviour of surrounding components is accurately specified, formal verification of a storage aplication directly addresses the problem of bugs causing corruption, however it incurs

significant engineering costs. In addition to the cost of specification and verification, formal verification requires reimplementing the storage system from scratch. Furthermore, current verification techniques are unable to handle today's concurrent, highly-optimized storage systems, since they result in unacceptable performance tradeoffs. Another proposed approach is N-version programming, for instance EnvyFS, which stores each piece of data in three different file systems [5]. This approach is effective, but it has high storage costs, and limits functionality to the least common denominator.

Instead of full, formal verification, this thesis proposes addressing the problem of data-corrupting bugs with *run-time invariant checking*. The goal of run-time checking is to detect incorrect behaviour as soon as possible, by checking invariants that circumscribe permissible states or state transitions. Early detection of failures reduces the need for recovery from backup by protecting good state from being overwritten by bad, and simplifies root cause analysis by limiting the scope of possible operations that could have caused the invariant to be violated. At the point when failure is detected, we have knowledge of the execution context in which the write occured; perhaps we know which user operations executed recently and can determine which ones were responsible for the offending change. The invariant-based approach is more flexible than a cumbersomely-specified, formally verified system, while still guaranteeing that the supplied invariants hold.

Our run-time consistency checkers can dectect corruptions that violate the *consistency properties* of storage systems. These properties are crucial, because they are the implicit pre- and post-conditions on the data for every operation. For example, if a program bug (e.g., a stray pointer, or a race condition) causes a corruption to a link in a singly linked list, the program loses list elements, and it may proceed to corrupt other arbitrary data. The standard for identifying file system corruption has been the file system consistency check (*fsck*) tool. Our run-time consistency checking systems have been designed to catch any corruption *at runtime* that fsck would have detected after the fact. Unlike an offline consistency check tool, this thesis describes a more efficient way to check consistency at run time incrementally, called *differential invariants*. Differential invariants use a comparison between the old and the new states in order to determine what needs to be checked, instead of scanning the entire persistent state.

Our approach to run-time consistency checking is not exclusive to file systems, but it is geared towards *storage* systems in general, because it leverages a feature already included in many storage systems called crash consistency. Crash consistency is the ability of the system to recover to a consistent state after a sudden power loss or operating system crash. It facilitates run-time consistency checking by providing a point in time when consistency invariants are *expected* to hold. Crash consistency mechanisms also supply important prerequisites for differential invariant checking by allowing us to identify what state recently changed at a point when we can access both the old and new values of updated fields.

In addition to checking the consistency properties, we also develop run-time invariant checks for the *atomicity* and *durability* properties of the crash consistency mechanism that are necessary to ensure that consistency is preserved through a crash or power loss event. Atomicity is the property that the system will be able to roll back to a consistent state on a crash. Durability is the property that once data has been acknowledged as written a system crash does not cause it to be rolled back or overwritten.

Checking consistency, atomicity, and durability does not provide as strong a guarantee as verified correctness; for example, a system is still *consistent* if the user requests one operation and the system instead performs another, different, operation. On the other hand, many of the data-corrupting bugs analyzed in a study of bugs in Linux file systems [52] violate one or more of the properties that we can check. Many common pointer and concurrency errors naturally lead to violations of consistency, and misuse of subtle APIs for crash consistency lead to violations of durability or atomicity.

The advantage of run-time checking, in contrast to static proofs of correctness, is that we can check a simplified model of correct behaviour, according to the invariants being checked. In contrast, static verification depends on all aspects of the system being correctly and fully specified, and incurs a high engineering cost.

Additionally, because the consistency, atomicity, and durability properties must hold regardless of the sequence or interleaving of operations, run-time checking can handle today's highly optimized, concurrent systems which are not yet amenable to formal analysis. Independence from the particular sequence or semantics of operations also means that all of these properties can be checked while keeping the run-time checker isolated from the storage application, providing stronger reliability guarantees; our run-time checking system does not need to intercept user inputs, or even access the volatile state of the system. Instead, the invariant checking system only needs access to storage itself and the I/O traffic between the application and storage.

These advantages enable us to apply our approach directly to *real-world* storage applications. By focusing on consistency, durability and atomicity, we avoid the complexity of full program verification, while still ensuring that a misbehaving system will not leave persistent data in a dangerous, inconsistent state. In this thesis, we have implemented run-time consistency invariant checks for the Ext3 and Btrfs file systems, and for a version of the Redis key-value store adapted for persistent memory. We have also implemented atomicity and durability checks for Ext3 (journaling) and Btrfs (copy-on-write). We show that all of the properties checked by an offline checker can be efficiently checked at run time.

1.1 Scope

We use run-time checking to ensure that updates to persistent data are consistent, and then subsequently to ensure that the consistent state is stored atomically and durably, even when the storage application is malfunctioning. This section describes the three types of property that this thesis is concerned with checking: consistency, atomicity, and durability.

Our concern is protecting data from corruption originating in *software*, or the volatile portion of memory, above the level of the storage device. Following the end-to-end principle of data integrity, an application that wishes to protect itself from *hardware* corruption should encode and replicate its data in a fault-tolerant manner [86]. Throughout the remainder of this thesis, we assume that either hardware is reliable or the storage application has been designed to tolerate hardware failures.

In our run-time checking systems, we check properties relating to the *consistency*, *atomicity*, and *durability* of storage applications. The acronym ACID (Atomicity, Consistency, Isolation, Durability) is frequently used to describe properties of transactional database systems. The *isolation* property means that operations in one transaction do not see the effects of operations in concurrent transactions, which may abort without becoming durable. Many storage systems (including file systems) do not support isolation, so while failures of isolation may lead to consistency violations (which we can catch), we do not check any isolation-specific properties. We adopt the term "transaction" to describe a sequence of writes that are atomic and become durable at a distinguished *commit* point, but unlike database transactions, aborts are not supported, and the effects of concurrent operations may not be isolated from each other.

Consistency is the guarantee that the data reflects some valid sequence of atomic operations on the data.

For example, a pointer that points to a region of unallocated memory violates consistency because no completed and correct program operations should result in that state. Furthermore, bugs in the program could lead to a premature `free()` or a stray write corrupting the pointer, leaving the program in an inconsistent state. The

corruption may not even be detected the next time the pointer is used; instead the program will read garbage data or overwrite some other structure, further propagating the damage.

Another way that consistency can be violated is due to a failure of concurrency control. If two operations simultaneously try to update a variable non-atomically, one of the updates may be overwritten. This will violate consistency if there is a relationship between the quantity being updated and another quantity elsewhere, for instance, the relationship between the number of free blocks of storage and the number of blocks claimed by files.

Atomicity ensures that the user never sees the effects of an incomplete operation. Sequences of writes which are meant to happen atomically can be interrupted by a restart or crash, resulting in a partially completed operation; this loss of atomicity could result in a violation of consistency. To overcome this, many storage applications (particularly file systems and databases) ensure atomicity in the case of failure using a crash consistency mechanism. Some storage systems provide full atomicity while others provide atomicity only for *metadata*. We describe two approaches to crash consistency, journaling and shadow paging, in Section 2.3. Bugs in these mechanisms may lead to violations of atomicity, exposing the system to corruption if a crash occurs.

Durability is the distinguishing characteristic of a storage system: it is the property that ensures an application can read back something written, even after the application or entire system is restarted. Because of the performance characteristics of the underlying hardware, there is often a gap in time between when an application writes some data and when it is durable. In order for an application to be assured that the data it wrote has reached the point of durability, it issues a command (often called *sync*, *flush* or *fence*), which returns only after the underlying layer reports completion. For instance, a user-space application may call fsync() on a file when it needs to know that the contents have been stored durably before reporting to the user that their document has been saved.

Durability is violated if data that has been reported durable disappears. For example, if a call to fsync() failed to flush a block to the disk due to a bug and the system subsequently crashed, the block may still contain its old contents. Another way that durability can be violated is if there are bugs in the crash consistency mechanism. For example, if blocks are written to a journal in the wrong order, it is possible that garbage data will be used to overwrite good data while recovering from a crash.

1.2 Run-time Invariant Checking

In this section we describe the approach we take to run time checking, at a high level. We do this by answering three questions: *When* should we check, *What* do we check, and *How* do we check it.

1.2.1 When to check

A key challenge of run time checking is one of timing, or "when to check" the invariants. Consistency invariants may be temporarily violated mid-operation. For example, while a node is being inserted into a doubly-linked list, the `next` and `previous` pointers may not match up until the insert is complete. While this is happening, the list is inconsistent, but this isn't abnormal or erroneous behaviour. It is important, then, to distinguish between points in time when the data *should* be consistent and when it may be (temporarily) inconsistent. We call the former *consistency points*, and when the program we are checking reaches a consistency point, it is safe to check consistency invariants.

We know that crash consistent storage systems must have consistency points because when a crash occurs, it is essential that the system can be restored to a consistent state before beginning any subsequent operations. At

any point in time it must be possible to either complete operations, or revert incomplete operations, using only what has been written so far. At some point there is a write after which an atomic operation will certainly be durable, even after a crash. This distinguishing write is called the *commit* and signifies a consistency point.

The ideal time to check the consistency of a transaction is immediately before commit. At this point, we have complete information about the contents of the transaction, and we know that it is supposed to be consistent. However, it has not yet become durable, so if the consistency check *fails*, we can prevent the commit from being written and thus prevent the corrupt transaction from becoming durable.

Uncommitted writes preceding a consistency point are flushed to storage before the commit, but in a crash consistent system they must be written to designated locations that are guaranteed not to overwrite data irreparably. Atomicity invariants check this process, ensuring that nothing is written that can't be undone. Durability invariants ensure that blocks have been correctly flushed to their final destination at the appropriate time, and that the commit process is durable.

Unlike consistency invariants, atomicity and durability invariants need to be checked on every write to storage. The reason for this is that *any* write could be destined for an incorrect location, overwriting durable data or interfering with crash consistency. While consistency invariants do not have to hold true for uncommitted writes, atomicity and durability invariants are necessary to ensure that uncommitted writes are not durable and that committed writes are.

Because the atomicity and durability properties concern the ordering of writes and flush operations, they can't be checked after a crash by an offline checker. Instead, they may manifest as violations of consistency if part of an atomic operation has been lost or overwritten. By checking every write, we ensure that even transient violations of atomicity or durability can be caught, and that we can distinguish between malfunction in the crash consistency mechanism and the rest of the storage application.

1.2.2 What to check

The properties we check are consistency, atomicity and durability as described in Section 1.1.

Consistency is determined by storage-application-specific consistency invariants. Consistency invariants are specified in terms of the contents of the application's data structures. For example, it is a consistency invariant of the Ext3 file system that if a disk block belongs to a file, then its address must be marked in a bitmap to indicate that it is allocated. A disk block *belongs to* a file if one of the pointer fields in the file's inode contains the address of the block, and so a specification of the consistency invariant relates the value of the pointer fields in the inode to the contents of the block bitmap.

The consistency invariants are the same global properties that an offline consistency checker would check. Consistency check tools exist for most major file systems (e.g., *fsck* and *CHKDSK*), as well as some databases and other programs (e.g., Microsoft Exchange). They are used in case of disaster to detect and attempt to repair a corrupt system. Consistency invariants may be derived from an existing offline checker, from documentation and specifications of the data format, or from an examination of the system's code. Invariants can be expressed in a query language like SQL for clarity [33]. Our experiences back up the utility of clear, declarative invariant specifications.

The atomicity and durability invariants are determined by the semantics of the crash consistency mechanism being used (as described in Chapter 5). Like the consistency invariants, they are also defined in terms of concrete data structures, but unlike the consistency invariants they have a temporal aspect as well, since they concern the ordering of writes and not just their contents. We call atomicity and durability invariants *location invariants*, because together they determine if it is permissible to update a particular storage location at the current time. For

instance, in a copy-on-write system used by Btrfs, one location invariant is that all writes must go to unallocated blocks, except for the write to the superblock which commits the transaction.

1.2.3 How to check

A key problem with offline consistency checking is that it is *global*, and thus scales poorly with data size. For instance, in a consistent Ext4 file system, if there is a pointer from an inode to a block, then that block should be marked allocated in the block bitmap. In order to verify that a single bit in the bitmap is correct, an offline checker must scan *all pointers* in all inodes on the disk, because any one of them could be pointing to the corresponding block. In contrast, our approach avoids global checking and focuses on checking updates to storage. This is possible because we have extra information available to us at run time, which enables us to use *differential invariants*, resulting in drastically better scalability with respect to total data size.

Differential invariant checking is a form of incremental checking, using knowledge of both the "old" and the "new" states in order to minimize the amount of checking that must be done. Differential invariants delineate the subset of state that must be checked after an update is performed on some known-consistent state, in order to verify that the updated state is still consistent. This takes advantage of the fact that we have already checked the previous durable state, relying on the assumption that hardware errors can be detected by existing reliability techniques. Instead of checking the consistency invariants directly, we check the differential invariants that apply to the updates that are about to become durable.

Differential invariant checking is only effective if we know everything that changed during an update. Because we do not trust the application to be correct, we must *intercept* writes to storage in a way that a buggy application will not circumvent. For file systems, we interpose on the block layer, either within the operating system or by posing as a remote block device using a hypervisor[1]. For applications using persistent memory, we use an existing persistence framework[2] which performs *instrumentation* of the application binary at compile time in order to record all writes. Additionally, we do not trust any of the volatile (i.e., in RAM) state of the storage application. Because the consistency invariants apply to the persistent data alone, our run-time checking systems do not need to even access the volatile portion of application state.

Since invariant checks are performed at the storage layer, they must be able to identify the relevant fields of data structures being updated in order to perform checking. We refer to the mapping between physical storage locations and the logical identities of structures and their fields as the "interpretation" of the data. Interpreting writes to the persistent data requires introspection, which is data-structure specific code that can determine the types and names of fields within the data structure.

In the Recon system we require file-system specific code to *infer* the type of a block based on pointers within previous blocks that have been read. This makes use of an assumption we call *pointer before block*: Storage systems access blocks by following pointers from a root (often called the super-block), and these pointers determine how the destination block should be interpreted.

In Ingot, our system for run-time checking in persistent memory, we ask the user to *annotate* their memory allocations with a type. This is simpler than inferring types based on pointers, and takes advantage of Ingot interposing on persistent memory allocation. When a write occurs, Ingot uses the address to determine the allocation that contains the write, and uses the type annotation to determine the type of structure that was updated.

[1] For more details on the role of the block layer, see Section 2.1

[2] For more details on persistent memory, see Section 2.2.

1.3 Thesis Contributions

This thesis demonstrates that we can protect storage systems from bugs that violate the consistency, atomicity, or durability properties of the system, and that we can do this at run-time. In contrast to previous work on consistency checking in storage systems, we detect corruption before it becomes durable.

We show that global consistency invariants can be checked efficiently if they are rewritten as differential invariants, which are triggered by updates to data structures. This approach combines techniques from database systems [61] with the insight that crash consistent storage systems allow us to examine both the old and the new states in order to determine precisely what structures were updated, created, or destroyed. Further, we show that differential invariants can check properties that otherwise require a complete scan of a large data structure, which is a limitation of offline consistency checking.

We demonstrate the effectiveness of our approach by implementing differential invariant checking in two systems. We have written invariant checkers for the Ext3 and Btrfs file systems using Recon, a framework we developed for run-time checking storage systems that use the block I/O interface. We have also developed the Ingot framework for systems using byte-addressable persistent memory, and written invariant checkers for a red-black tree and a version of the Redis key-value store which we adapted for persistent memory. The invariants catch a wide variety of injected faults before the corruptions become durable.

We show that location invariants can be used to check the atomicity and durability properties of a crash consistent storage system. We describe the location invariants for two systems in detail: the JBD journal used in Ext3 (a redo log), and the copy-on-write B-trees used in Btrfs (shadow paging). We evaluate the effectiveness of the invariants through fault injection, and also describe how file system design affects the feasibility of checking location and consistency invariants.

1.3.1 Thesis Organization

In the next chapter, we give some background information on storage systems relevant to this thesis. In Chapter 3, we describe related work relevant to the problem of protecting data in storage systems. In Chapter 4 we describe Recon, a system for checking the consistency of a file system's writes at run-time. In Chapter 5 we describe the *location invariants* used by Recon to further ensure that consistent updates are made durable in a failure-atomic manner. In Chapter 6, we describe Ingot, a system for checking that an application's data structures are consistent in byte-addressable persistent memory, using principles developed during the Recon project. Chapter 7 describes opportunities for future work, and Chapter 8 summarizes and concludes the thesis.

Chapters 4 and 5 of this thesis are based on material previously published by the Association for Computing Machinery [25, 26].

Chapter 2

Background

This chapter describes the components of a storage system, how they interact, and how they relate to our implementations of run-time invariant checking. This thesis describes checking systems for two very different kinds of storage devices, block storage (e.g., hard disks) and byte-addressable persistent memory (e.g., phase change memory).

The Recon system was designed to check storage systems built on top of block storage devices. We will describe block oriented storage systems in Section 2.1, and then explain how new byte-addressable persistent memory devices alter the assumptions made about storage in Section 2.2. Because persistent memory offers a fundamentally different interface to storage, as well as very different performance characteristics from block devices, applications need to be modified to take advantage of it. We built the Ingot system to check the consistency of data being stored by these modified applications which bypass the block layer.

Our approach to run-time consistency checking requires that the application being checked is crash consistent. Section 2.3 describes journaling and shadow paging, two different techniques for achieving crash consistency that are relevant to the Recon and Ingot systems.

2.1 Block Storage

The majority of storage systems in use today are built around the *block storage* abstraction. This category includes magnetic hard disk drives and flash-based SSDs. The block storage abstraction represents a device as a linear array of blocks or *sectors* of a fixed size, often 512 bytes (in older devices), or 4096 bytes (in newer devices). Requests to read and write must be sector-aligned and in multiples of the sector size.

In this section we describe three important layers of the storage stack (Figure 1.1) starting from hardware at the bottom, followed by the block layer of the operating system, and finally file systems, which are the primary storage application we are concerned with checking.

2.1.1 Block Storage Devices

At the lowest level of the storage stack is physical storage media, whether magnetic (as in hard disks), capacitative (as in flash), or optical (as in CDs and DVDs). Because this layer is subject to physical interference, data is usually encoded in a form that allows the storage device to correct minor media errors (on the order of a few bits) and to detect major media errors, which are reported to the layer above as a read error.

Writes issued to a storage device are often not immediately durable, because the order in which requests are processed has a large impact on device performance. Most storage systems include some amount of volatile cache to buffer requests so that the controller can order them more optimally. While a write request will be acknowledged as soon as it reaches the device cache, it is not guaranteed to reach the storage media in the event of a power failure. In order to know that data has reached persistent storage, it must be *flushed* by a command to the device. The ordering of these flush commands with respect to other writes is critical to the atomicity and durability invariants that we define and check in Chapter 5.

To help higher-level storage systems cope with power loss events, storage devices usually guarantee the atomicity of writes at sector granularity. Even when the power fails, a sector-sized write should happen completely or not at all. Error correcting codes internal to the storage device are used to detect a failure of atomicity at this level, and reported as a read error.

2.1.2 The Block Layer

The portion of the operating system responsible for interacting with block devices is called the *block layer*. It serves as an abstraction layer between the devices and the file systems and applications that use them for storage.

Within the Linux operating system, the interface to the block layer is the block I/O request, or *BIO*. A BIO is an asynchronous request to read from a device into a buffer in volatile memory, or write the contents of a buffer to a device. Each BIO has a *header* which identifies the block device and address being accessed, as well as flags which indicate whether the request is a read or a write, and if it has any special status (e.g., if it requires a flush to the device, or resulted in an error).

Because the block layer interface is a very simple abstraction, it is not only used for local, physical devices but also for virtual or remote storage devices. For instance, multiple disks may be joined together in a logical volume, which appears as a single block device to the layers above.

2.1.3 File systems

Most storage is accessed through a file system, which organizes raw storage into files and directories. The file system is responsible for managing metadata which is used to locate files, keep track of file attributes like ownership and permissions, associate file offsets with addresses on the disk, and keep track of allocated space.

Because the integrity of this metadata is crucial to the operation of the file system, some file systems take measures to protect their metadata from corruption originating at the hardware level. For instance, both the Btrfs and ZFS file systems use checksums to verify the integrity of a block containing metadata when it is read, and store each metadata block in two separate locations so that it can be recovered on corruption. This protects against corruption that occurs after the checksum is calculated, but not against corruption that happens in memory, before the checksum is calculated.

File systems frequently need to perform atomic operations that involve multiple writes, in order to update data and metadata in multiple locations. For example, when you create a new directory, the file system might create an *inode* structure for the new directory in one place, and a *directory entry* structure in another. Because atomicity at the device level is usually limited to a single small write, sequences of writes which are meant to happen atomically can be interrupted by a restart or crash. This loss of atomicity (completion of some, but not all, of the writes) corrupts the file system. This is why file systems (as well as other storage systems, like databases) employ additional mechanisms for crash consistency. Crash consistency is also called *failure atomicity*, referring to the preservation of atomicity in the presence of power or crash failures.

There are many different file systems, designed for a variety of hardware, use cases, and operating systems. Originating in Sun UNIX+, many operating systems provide an abstraction called the Virtual File System (VFS), which unifies different physical file systems under a common name space and interface [47]. User applications interact with VFS through system calls (open, mkdir, write, read etc.) which then dispatches the request to the appropriate file system. To simplify the efficient implementation of file systems, operating systems also provide a *page cache*, a centralized mechanism for the contents of block storage devices to be cached in memory. Because block storage devices are typically much slower than main memory, caching is essential for good performance. File systems use this cache to buffer updates as well as to keep contents of recently read files available. Because of caching at the system and device level, applications must call fsync() or sync() after modifying data or metadata in order to ensure that data is durable.

2.1.4 Checking invariants for block storage systems

The Recon system for run-time invariant checking is positioned within the block layer, where it can intercept traffic between a file system and the disk. By posing as a block device, Recon is transparent to the file system; after receiving a block I/O request it forwards the request to an underlying storage device. Because Recon intercepts all I/O between the file system and storage, we can guarantee that we can catch writes which violate the invariants. From within the block layer Recon is also able to cache metadata and issue its own read requests in order to retrieve the consistent, durable state of the file system, without accessing the page cache which may be undergoing modification from the file system.

Positioned below the file system, Recon does not see which VFS operations are supposed to be executed, but since consistency is independent of the sequence of operations performed, this does not matter. By isolating Recon from the filesystem, bugs in the file system can't cause Recon to bypass invariant checking. For further isolation, Recon can pose as a block device on a *remote* machine, either across a network connection or via a hypervisor on the same machine. This gives Recon isolation from misbehaviour in the target file system and the OS that hosts it.

2.2 Persistent Memory

This section describes a relatively new type of storage called persistent memory, also known as Non-Volatile Memory (NVM). Persistent memory is distinguished from block storage by one key property: it is connected to the address bus, and so it can be directly read and written with load and store instructions. This enables new low-latency storage applications which do not need to invoke the OS to perform reads and writes, once the persistent memory has been mapped into their virtual address space. Rather than writing or reading an entire 4k block at once, access happens at the cache-line granularity.

Without a block layer handling I/O between the application and storage, and without access divided on block boundaries, applications can bypass an operating system component like Recon. Instead, we have designed Ingot to be integrated into an application alongside a *persistence framework* which uses binary instrumentation to ensure that all persistent writes are logged. The purpose of the persistence framework is to provide atomicity and durability to the application using persistent memory.

The following subsections give an overview of persistent memory hardware, the role of the operating system in managing persistent memory and how it differs from its role in block storage, and finally a description of why and how persistence frameworks help developers writing applications for persistent memory.

2.2.1 Devices

There are several competing technologies in the persistent memory space that use different physical characteristics to encode storage, including phase-change memory (PCM), spin-transfer torque memory (STTM), and resistive RAM (ReRAM) among others. Their common feature is that they all allow reading and writing at a granularity of individual bytes. They are also much faster than NAND flash memory, with read and write latencies expected to come within a factor of 10 of DRAM. Device density and cost is projected to fall in between flash memory and DRAM.

This combination of low latencies and fine-grained updates makes it feasible to attach these devices directly to the memory bus like DRAM. Unlike DRAM, however, they retain their contents after a power failure. The tricky part is that the durable contents of persistent memory may differ from the application's current view of memory, due to caching in the CPU. When writing to a persistent region of memory, writes are cached within the CPU (L1, L2 etc.). These updated cache lines may be flushed in arbitrary order, similar to how the contents of a disk cache may not become durable in the order they were written.

Control over the order in which writes become durable is important for maintaining consistency; if one operation depends on a previous operation, its writes should not become durable until the previous operation has become durable. Most architectures have instructions dedicated to flushing the CPU cache, and new instructions are being added to support persistent memory-specific flush operations, like writing a cache line back to memory without simultaneously evicting it from the CPU. These instructions, in conjunction with memory fence instructions, can be used to control the order in which writes appear in persistent memory.

In order to insulate application programmers from the subleties of using persistent memory in a crash-consistent, durable fashion, a number of different frameworks and APIs have been proposed for using persistent memory, which we refer to as persistence frameworks.

2.2.2 Persistence frameworks

Persistence frameworks serve as a layer of abstraction on top of the hardware-specific details. They provide interfaces for controlling the ordering of writes, ensuring durability, and allocating persistent memory. They also provide APIs for crash consistency. Several different frameworks have been proposed to make it easier for applications to take advantage of persistent memory. They provide support for crash consistency in one or more ways, whether it is in the form of explicit logging [41, 82], a persistent object API [14, 41, 56], an abstraction like software transactional memory [51, 82], or by instrumenting existing lock-based concurrency control [10, 40].

Using explicit logging or a persistent object API requires rewriting significant parts of the application in order to fit the programming model used by the persistence framework, but has the benefit of potentially higher performance. Approaches based on software transactional memory or lock instrumentation can be used with fewer changes to the application, but exhibit higher overhead.

2.2.3 Operating system management

The role of the operating system in managing persistent memory is a hybrid between its role in file systems and in virtual memory management. The operating system is responsible for naming and allocating tracts of persistent memory to an application, and mapping the designated persistent memory into the application's address space on demand. It then becomes the application's responsibility to allocate persistent objects within that region and ensure that it can recover from a crash.

It is also possible to use persistent memory through a file system (using either read/write or mmap system calls), but in order to respect file system durability semantics the application must call fsync() or msync() before it can expect data to be durable. This forgoes some of the latency benefits of persistent memory, in exchange for a more familiar file-based interface.

2.2.4 Checking invariants in persistent memory

Because in-memory reads and writes don't go through the operating system, the approach we use in Recon to interpose on block storage devices does not apply to persistent memory applications. In Ingot, our system for run-time checking in persistent memory, we use an existing persistence framework called ATLAS [10]. The goal of ATLAS is to make it easy for existing lock-based applications to use persistent memory. ATLAS provides a persistent memory management API, and transparently provides crash consistency by instrumenting the application at compile time. This is similar to some implementations of software transactional memory, except that ATLAS uses the existing lock-based concurrency control of the application to infer transaction boundaries and relies on the application's own concurrency control instead of performing conflict detection. The advantage of using a tool which instruments all stores is that we can catch all persistent writes, even those which result from a corrupted pointer intended for volatile memory.

2.3 Crash consistency

Crash consistency is the ability of an application to return to a consistent state after a crash, preferably one which reflects the most recent consistent state before the crash. Because operations which should happen atomically often involve multiple writes, some mechanism is needed to ensure that either all or none of the writes become durable. A group of writes becomes durable when a particular write called the *commit* becomes durable. A single commit may cause many atomic operations (hundreds, or thousands) to become durable together as part of a large transaction; this is called *group commit* [36]. In this thesis, we do not distinguish between the commit of a single operation and a group commit, because consistency invariants apply to any composition of operations.

There are two main crash consistency mechanisms in use by common storage systems, journaling and shadow paging (copy-on-write), described below. Other mechanisms which provide weaker guarantees than full atomicity and durability after a crash have also been proposed [13, 27].

Most file systems protect their metadata using some form of crash consistency, but because of the performance overhead of crash consistency mechanisms, some file systems make it optional to supply the same guarantees about user data in files. This practice is justified by increased performance and the argument that applications are responsible for the crash consistency of their data. When the file system does not provide larger atomic writes, applications can only rely on atomicity at the granularity of the underlying device, which may not be possible to establish from userspace. User space applications implement their own journals or use the atomic `rename()` system call to ensure crash consistent data.

2.3.1 Journaling

Journaling (also called write-ahead logging, WAL, or just logging) is used by file systems and databases to provide crash consistency (e.g., Ext3 [81], XFS [78], or NTFS [18]). A journal can be used in two different ways: to undo an incomplete operation (called *undo logging*) or to finish an incomplete operation (called *redo logging*). In either case, there is a designated area of persistent storage, called the journal, which is used to record enough information

to restore a consistent state if a crash occurs. Journals can be *logical* (recording what operation was performed and any parameters necessary to undo or redo it) or *physical*, recording the writes to be performed (physical redo logging) or undone (physical undo logging).

Journaling file systems primarily use physical redo logging. A system using physical redo logging first writes a copy of all of the updates in a transaction to the journal, along with metadata indicating where the writes should take place. Once all the writes have been made durable in the journal, the transaction is committed by a single atomic write called a *commit record*. The commit record indicates that the previous group of writes is complete, and once the commit record has been flushed, it is safe to start writing the blocks in the journal to their final destinations. If a crash happens, then during recovery the presence of the commit record indicates that the preceding group of writes should be re-done in case they were interrupted, hence *redo* logging. Crash or not, once the last write has been copied to its final destination, the portion of the journal containing the commit record and its preceding writes can be freed and reused.

Physical redo logging is useful for run time consistency checking, since the journal contains the new state needed for differential invariant checking, while the final destinations indicated in the journal metadata contain the old state that is about to be overwritten.

It is also possible to use physical undo logging for invariant checking. Ingot uses the ATLAS persistence framework, which keeps an undo log. In undo logging, the journal contains the contents of areas that are about to be overwritten. If a crash occurs before the commit record is written, then all of the old values are copied to their original location, to undo any partially completed updates. If there is no crash and the operation finishes, a commit record is atomically written, which indicates that the preceding log entries should *not* be used to undo what is now a complete operation.

In contrast, the high-level changes recorded in logical journals need to be materialized as changes to the underlying data structures before their consistency can be checked. For instance, if an entry in a logical journal specifies that a new directory ”Foo” should be created inside directory ”Bar”, this does not provide us any information about whether the file system will alter the directory data structure for ”Bar” *consistently*. If the physical materialization of the logged updates directly overwrites the previous state, then by the time the complete new state has been written, the old state will not be available for differential invariant checking.

2.3.2 Copy on write

Another crash consistency technique is called shadow paging (or *copy-on-write*), used by file systems like ZFS [7], Btrfs [68] and WAFL [38]. Shadow paging systems organize their data into a tree, with a root that is pointed to by a special block location called the superblock. In between the root and the tree leaves which contain the data are a hierarchy of index pages, which map logical addresses to physical addresses. Rather than writing everything twice (once to the journal, and once to the final destination), the principle of shadow paging is to first write new or updated versions of pages to unallocated space (including updated index pages) and then perform an atomic overwrite of the superblock to point to the new root. Overwriting the superblock is the commit point in a copy-on-write system, and can only happen after the rest of the updates to the tree have been flushed to storage.

Log-structured file systems are a special case of copy-on-write file systems which aim to keep their writes contiguous in order to maximize device performance. Log-structured file systems work similarly to other copy-on-write systems with respect to crash consistency, with a superblock that points to the tail of the log being updated.

2.3.3 Persistent Memory

The techniques used to achieve crash consistency in persistent memory are broadly similar to those used in file systems and databases designed for block storage, except that writes are performed at cache-line granularity (32-64 bytes) instead of block granularity.

Storage systems designed for block storage typically buffer writes in volatile memory before sending them to disk. This gives the operating system (and hence the file system and block layer) control over when writes occur. In contrast, writes to persistent memory become durable when the CPU flushes the updated cache line to the device. This may happen as the result of an explicit cache flush instruction, but it also may happen at any time that the CPU decides to evict the cache line. Since this is not under the control of the application or operating system, applications written for persistent memory must be careful about the order in which they assign values to fields in persistent structures. This is one reason to use a persistence framework which hides some of these details from the application writer.

2.4 Fault Tolerance

Fault tolerance in storage systems has primarily focused on failures at the device level, because device-level failures have historically been where most corruption came from. However, there are a number of well-studied and commonly implemented techniques for dealing with failures that occur below a file system. Additionally, the late detection of storage corruption has meant that corruption due to software problems (e.g., a dangling pointer leading to the overwriting of some metadata) was hard to distinguish from corruption due to device problems (e.g., the same metadata being corrupted by physical means).

There are many possible causes of corruption below the file system, including:

- damage to the physical media,
- (physical) failure to write some data
- physical interference during writes to adjacent data
- errors while in a device's volatile cache
- controller errors causing a write to an incorrect location
- controller errors dropping a write entirely, or reporting durability prematurely

Fault-tolerant hardware systems (e.g., RAID) have been designed to mitigate device and media failures by replicating data across multiple devices. This is helpful not only for detecting and recovering from corruption, but also from total device failure (e.g., a head crash on a hard disk drive, or a broken connector). Mirroring refers to the storing of multiple whole copies on different devices. A more space efficient approach is to use erasure codes to distribute the data across multiple devices while still tolerating one or two complete device failures. Erasure codes enable a tradeoff between space efficiency, computational efficiency, and device bandwidth.

File systems can also tolerate lower level failures as long as they can detect corruption and keep multiple copies of crucial metadata. By keeping a checksum of every block, a file system can detect whether the underlying device(s) are returning what was expected [7]. This also provides some protection against corruption due to bugs in the block layer or device drivers, but not in the file system itself.

Hardware fault tolerance has the advantage of being able to use extra hardware resources to ensure that the burden of fault tolerance does not adversely impact performance. In fact, when using techniques like battery backed RAM in a RAID array, performance increases because flushed writes can be acknowledgd as durable as soon as the write arrives in device memory, instead of waiting for it to reach the storage media.

In contrast, software techniques can take advantage of specific knowledge about the data being stored. For instance, they can avoid reconstructing copies of unallocated blocks after a crash, and they can selectively choose to replicate data according to its importance. For this reason, some file systems have begun to manage pools of disks, similar to a hardware RAID array, in order to perform their own mirroring or erasure coding.

The Recon system checks the consistency of writes coming from the file system, and leaves the responsibility for tolerating lower level failures up to the file system and existing devices. This complements existing fault tolerance mechanisms; while tolerance of lower level failures seeks to preserve what the file system wrote, Recon ensures that what is being preserved is itself consistent.

Fault tolerance for byte-addressable persistent memory is still in its infancy, as persistent memory devices are only recently becoming available on the market and their failure characteristics are not yet known. Persistent memory chips can use error correcting codes, but they may also be subject to wear and burnout like NAND flash, necessitating wear levelling or substitution of one page for another.

2.5 Summary

In this chapter we present important background material. Traditional storage media is accessed through the block layer in the operating system, on top of which storage applications like file systems and databases are built. Persistent memory can be read and written without operating system intervention, but persistence frameworks have been developed in order to ease the task of writing fault-tolerant applications. Crash consistency is an important characteristic of storage systems, and is usually accomplished by techniques based on journaling or shadow paging.

We also previewed some of the ways that these concepts are relevant to run-time consistency checking for both traditional block storage and new persistent memory. In the next chapter, we review closely related work on preventing corruption.

Chapter 3

Related Work

This chapter describes other work which attempts to prevent or repair data corruption and how it relates to this thesis.

3.1 Eliminating Bugs

The ideal time to prevent data corruption is before it happens in the first place. There are several approaches to bug-finding in general which are applicable to bugs that corrupt data.

3.1.1 Static Analysis & Symbolic execution

Bug finding tools, based on model checking [85] and static analysis [69], have revealed scores of bugs in a variety of file systems. However, these tools cannot be relied upon to identify all bugs because they need to perform exhaustive evaluation.

Symbolic execution has been used to explore whether certain inputs can drive the file system into an undesirable action (e.g., a null pointer dereference or an assertion failure) [8, 83, 84]. This is useful for hardening the read path protecting the file system's execution and in-memory state from a corrupt disk, but not for protecting the stored data from a misbehaving file system. In conjunction with a run-time consistency checking system, these approaches could be used to find a consistent input file system that leads to an inconsistent output file system.

Building formally verified software, in particular file systems, has been an attractive goal for verification [43]. The FSCQ file system [12] has been provably certified to provide crash safety. The underlying Crash Hoare logic (CHL) used by FSCQ allows developers to prove specifications about crash consistency. Another filesystem, Yxv6, has been verified using a technique the authors call crash refinement [72]. It uses a specification of correct behaviour, the data layout, and consistency invariants, in order to prove whether or not an implementation meets the specification.

The consistency invariants that we check ensure that the data is structured correctly, but they do not ensure that operations on the data structures perform the desired behaviour. Our approach imposes significantly less burden on programmers for specifying the application's consistency invariants. Additionally, we can apply run-time checking to existing highly-optimised file systems without modifying the file system itself. Current approaches to verification are unable to handle some important techniques for high performance file systems, including multithreading.

3.1.2 Testing & Assertions

One way to look at run-time checking is as a set of assertions that execute after every atomic operation, independent of code path. Assertions help with writing test cases, enable systematic test case generation [8, 29], and are generally known to improve software quality and reduce bug density in production systems [48]. run-time checking can similarly be used during software development to help detect corruption bugs, and differential invariants make it possible to check global assertions efficiently.

Because different file systems present a similar interface, there are suites of file system tests designed to flush out corner cases that have been developed over the years [17]. This is helpful for testing in general, but it leaves the detection of corruption to an offline checker. If a corrupted state is generated but overwritten partway through the test, it won't be caught by a consistency check after the fact. Furthermore, these global consistency checks are expensive. Using run-time checking in conjunction with these test suites makes it possible to detect corruption as soon as it happens.

The problem of transient corruption is especially relevant when testing crash consistency code. The Yat tool [50] was designed specifically for testing the crash recovery mechanisms of file systems developed for persistent memory. It records the order of writes and flushes, simulates crashes by permuting and selectively discarding writes between the last two flush commands, and tests whether the file system can recover to a consistent state. Yat can test many permutations, but the space of possible orderings is too large to search exhaustively. Our invariant checking takes a different approach, using a logical model of when flushes *should* be seen and when writes to a particular location *should* be allowed.

Similarly, the CrashMonkey tool for *bounded black-box testing* of file systems [60] explores possible outcomes of file system writes with respect to durability, and then checks the consistency of the file system. CrashMonkey uses a small RAM disk to test the file system in order to accelerate the testing of file system configurations. CrashMonkey could be modified to use Recon-like atomicity and durability invariants, which would detect inconsistencies immediately without having to stop the file system and check consistency offline.

Valgrind is a tool that checks for dangerous behaviour in programs written in C and C++. In particular, it is frequently used to detect memory corruption stemming from buffer overruns, the use of uninitialized values, and the use of stray or dangling pointers. Valgrind can catch suspicious behaviour even if it doesn't violate a consistency invariant. However, it is not suitable for enforcing arbitrary invariants on data at scale, and often has a high false positive rate. This makes it suitable for improving code quality generally, but does not provide any guarantees about data integrity.

3.2 Offline Checking and Repair

Offline consistency checking tools were initially developed because early file systems were not crash consistent, and power failures or sudden device disconnection were not uncommon. After a sudden shutdown, a consistency check and repair tool had to be run on the volume in order to restore the file system to a consistent state, possibly losing some recent changes in the process. Conflict between two pieces of information are resolved in the most conservative way possible; for instance, in the e2fsck repair tool for ext2, Ext3, and Ext4, if two files claim the same block (which is forbidden in the ext file systems), then the shared block is copied and each file is updated to point to their own copy.

These tools also turned out to be useful for causes of inconsistency other than crashes. For example, Linux and Windows desktop systems automatically perform a consistency check on startup after a specified time period or number of power cycles has been passed. This helps to detect inconsistencies due to media errors, memory

errors or software bugs. However, offline consistency checking is slow, and the repair process is difficult because of the many possible combinations of ways that the consistency invariants can be violated.

Recently, there has been significant interest in improving the performance and robustness of offline consistency check and repair tools. The REXt3 file system [53] uses backpointers and collocates its metadata blocks, allowing its ffsck checker to scan the file system at rates close to the sequential bandwidth of the drive. Chunkfs [37] reduces the time to check consistency by breaking the file system into chunks that can be checked independently of each other. The SQCK offline consistency checker [33] expresses file system consistency properties declaratively, demonstrating that file system checks and repairs are more easily understood when expressed as SQL queries. It improves upon the repairs made by e2fsck by correcting the order in which certain repairs are performed and by using redundant information already provided by the file system. The SWIFT tool [9] tests the correctness of offline file system checker recovery code by leveraging the file system checker itself or by comparing the outputs of multiple checkers. The rfsck (robust fsck) family of tools was developed in response to the observation that power loss during repair could corrupt the filesystem uncorrectably [28], demonstrating the need for robust crash consistency mechanisms during the repair phase of a file system check and repair tool. The authors of rfsck use fault injection in order to demonstrate that the file system check tools for the XFS and the Ext4 file systems were vulnerable to interruption and that their robustness library is effective.

3.3 Runtime Checking

Our work is influenced by runtime verification, a technique that applies formal analysis to the running system rather than its model [11, 76]. Runtime verification encompasses many different approaches to specifying the properties and events being monitored. The monitoring oriented programming (MOP) framework allows a developer to define properties according to a logical formalism and then synthesizes a monitoring program. JavaMOP, an implementation of this idea, allows users to specify queries about the ordering of events, for instance, to verify that a user is authenticated before a resource is accessed. JavaMOP uses techniques from aspect-oriented programming to execute code before and after specified function calls. In contrast, our approach relies on intercepting flows of data between components of the system – specifically, between a storage application and the hardware that stores data persistently.

Our work on differential invariants is heavily influenced by differential relational calculus developed for efficiently checking database integrity [61], and maintaining materialized views [31, 35]. Our contribution stems from adapting these ideas for checking the integrity of non-relational, persistent data structures, that are updated concurrently, and in a non-managed environment.

3.3.1 In Storage Systems

Based on several requests, checks for location invariants and some consistency invariants were added to Btrfs as a debugging tool [6]. These checks catch common errors, but they are embedded within the file system code itself, and so, for example, a file system bug could disable them.

NetApp implemented runtime consistency checking for their WAFL file system [49]. Their checking infrastructure is deployed in the field and must be efficient, and so they keep the number of inter-structure integrity checks to a minimum. Programmers explicitly log updates so that the correct invariants are triggered, and a checksum mechanism is used to detect spurious overwrites. They observed that their system was effective in detecting bugs during development and after deployment.

HARDFS [21] detects software bugs in the Hadoop distributed file system (HDFS) at runtime by interposing on network messages and I/O, and verifies that the HDFS implementation behaves according to its operational specification. The verification state is compressed using bloom filters, significantly reducing the memory overhead. HARDFS can check certain end-to-end properties that a consistency checker cannot, such as whether a request was performed, but HARDFS does not attempt to catch all failures or guarantee that it will not raise false alarms.

3.3.2 In Memory

Our work on run-time checking for programs using persistent memory is closest to checking heap assertions in managed code environments [1, 66, 71]. Strobe [1] creates a consistent snapshot of a heap at object granularity, allowing assertions to be checked asynchronously in separate threads. DeAL [66] is a logic-based extension to Java that supports assertions on heap objects during garbage collection (GC). Invariants are checked at GC granularity, giving low overhead, but it is unlikely to catch problems as they occur. DeAL depends on the fact that a full GC will visit every node once, allowing efficient checking of invariants based on reachability, such as cycle detection and dominators in a graph. DeAL invariants are only permitted a single variable in their first-order logic specification, so that the invariant can be checked in a single GC pass.

The Ditto system allows users to incrementally check the integrity of heap objects in Java. The user specifies invariant checks as a recursive, side-effect-free function, which is responsible for traversing all relevant objects. Ditto memoizes in order to determine which portions of the global integrity check must be recomputed when the underlying data changes. This is an attractive, user-friendly approach to specifying invariants, but it imposes the memory cost of memoization of each recursive invocation of the function, and recording the associations between data elements and function invocations.

These systems leave several challenges unaddressed. Unlike Java, programs written in C and C++ allow pointer manipulation. This presents a challenge to a run-time checking system because pointer corruption can result in arbitrary memory corruption. Additionally, previous work does not handle concurrent updates, or assumes that the invariants can be coded to run atomically. Finally, while memoization and GC traversal both provide useful and interesting ways to incrementally check invariants, they miss out on efficiencies that could be gained by using both the old and the new state to check invariants, as described in Section 6.3.

3.4 Software Fault Tolerance

While rigorous testing and good engineering practices reduce the number of serious bugs in storage systems, they have not eliminated the problem entirely. The risk of adopting an "immature" storage system means that it takes years before a new file system is trusted. Even when a bug is found, a bug fix may not be easily available, or easy to deploy in live systems [3]. These limitations can be addressed by tolerating bugs at runtime.

There has been significant work on discovering program invariants by capturing variable values at key points in a program to repair data structures [20] and to patch buggy deployed software [62]. These systems may over- or under-generalize based on their observations of common patterns, resulting in false positives or negatives. Since storage system authors often explicitly specify invariants for offline consistency check and repair tools, it is more reliable to leverage this insight into correct system behaviour. On the other hand, using tools to discover invariants may be useful when developing a checking tool for an unfamiliar system that lacks clear format specifications.

EnvyFS [5] uses N-version programming for detecting file system bugs at runtime. It uses the common VFS

interface to pass each VFS-layer file system request to three child file systems and uses voting when returning results. The runtime overheads of this approach are high and subtle differences in file system semantics can make it hard to compare results. Additionally, this approach limits file system functionality to the lowest common denominator of the file systems used.

Membrane [77] proposes tolerating bugs by transparently restarting a failed file system. It assumes that file system bugs will lead to detectable, fail-stop crash failures. However, inconsistencies may have propagated to the on-disk metadata by the time the crash occurs. Our approach is complementary to Membrane, rather than waiting for the file system to crash, a restart could be initiated as soon as Recon detects an inconsistent transaction.

3.5 Monitoring and Introspection

Our checker leverages ideas from semantically-smart disks [75], which use probing to gather detailed knowledge of file system behaviour, allowing functionality or performance to be enhanced transparently at the block layer. I/O shepherding [32] builds on smart disks, allowing a file system developer to write reliability policies to detect and recover from a wide range of storage system failures. Unlike smart disks, a type-safe disk extends the disk interface by exposing primitives for block allocation [74], which helps enforce invariants such as preventing accesses to unallocated blocks. Rather than performing introspection to infer block allocations, type-safe disks present a richer interface so that the storage system can make useful information explicit.

The InkTag system is designed to protect a "high assurance process" (HAP) from a potentially malicious operating system [39]. InkTag itself runs in a hypervisor where it can restrict changes to virtual memory mappings. It also ensures that the untrusted operating system writes the correct data buffers to storage when the HAP writes data. However, InkTag does not manage file system metadata and instead keeps its own private file system metadata store. By using a system like Recon, which can interpret file system metadata updates, a monitoring system like InkTag could check that particular metadata operations make it to disk safely. To make checks like this, it is necessary to have a model which relates VFS operations to file system metadata operations; InkTag has such a model for data operations but could be augmented with a model of metadata operations.

3.6 Summary

Since the integrity of stored data is paramount to the notion of a storage system, many aspects of fault tolerance have already been considered. When it comes to bugs in storage software, there are several avenues that have been considered: static verification and testing (to eliminate bugs in the code), offline checking and repair (for disaster recovery), tolerance of bugs at run-time, and run-time checking to detect corruption. Our work falls into the latter category, but distinguishes itself by being designed to check all of the critical consistency, atomicity, and durability invariants of existing systems. These systems are typically written in C, and utilize multiple threads which makes them unsuitable for current static verification techniques. Additionally, our work aims to isolate itself from errors in the application which could compromise the checking process.

In the next chapter, we describe Recon, the first such system we built, designed to check file system consistency from within the block layer.

Chapter 4

Recon: Checking File System Consistency at Run-time

It is no surprise that file systems have bugs [64, 83, 85]. Modern file systems are designed to support a range of environments, from smart phones to high-end servers, while delivering high performance. Further, they must handle a large number of failure conditions while preserving data integrity. Ironically, the resulting complexity leads to bugs that can be hard to detect even under heavy testing. These bugs can cause silent data corruption [63, 64], random application crashes, or even worse, security exploits [84]. A comprehensive study recently showed that 40% of file system bugs have severe consequences, because they lead to in-memory or on-disk data corruption [52].

Unlike hardware errors and crash failures, it is much harder to recover from data corruption caused by bugs in file-system code. Hardware errors can be handled by using checksums and redundancy for error detection and recovery [7, 32]. Crash failure recovery can be performed using transactional methods, such as journaling [36], shadow paging [38], and soft updates [27]. Modern file systems, such as ZFS, are carefully designed to handle a wide range of disk faults [86]. However, the machinery used for protecting against disk corruption (e.g., checksums, replication and transactional updates) does not help if the file system code itself is the source of an error, in which case these mechanisms only serve to faithfully preserve the incorrect state.

File system bugs that cause severe metadata corruption appear regularly. To illustrate the problem, we examined bugs in two Linux file systems: the widely used Ext3 file system and the recently deployed Btrfs file system. We searched for "Ext3 corruption" and "Btrfs corruption" in various distribution-specific bug trackers or mailing lists. Based on the bug description and discussions, we removed bugs that did not cause metadata inconsistency, or were not reproducible, or were reported by a single user only. Table 4 summarizes the remaining bugs[1]. Note that Ext3, despite its maturity and widespread use, shows continuing reports of corruption bugs. These reports likely under-represent the problem because the bugs that cause metadata corruption may be *fail silent*, i.e., the error is not reported at the time of the original corruption. By the time the inconsistencies appear, the damage may have escalated, making it harder to pinpoint the problem.

When metadata corruption is discovered, it requires complex recovery procedures. Current solutions fall in two categories, both of which are unsatisfactory. One approach is to use disaster recovery methods, such as a backup or a snapshot, but these can cause significant downtime and loss of recent data. Another option is to use an offline consistency check tool (e.g., e2fsck) for restoring file system consistency. While a consistency check can detect most failures, it requires the entire disk to be checked, causing significant downtime for large

[1] As of Sept. 2011

Bugs causing data corruption in the file system (2011). All Red Hat and Debian bugs were rated high-severity. The severity level of bugs obtained from mailing lists is not known.

FS	Source	Bug Title	Closed
Ext3	http://lwn.net/Articles/2663/	Ext3 corruption fix	2002-06
Ext3	kerneltrap.org/node/515	Linux: Data corrupting Ext3 bug in 2.4.20	2002-12
Ext3	Redhat, #311301	panic/Ext3 fs corruption with RHEL4-U6-re20070927.0	2007-11
Ext3	https://lkml.org/lkml/2008/12/6/88	Re: [2.6.27] filesystem (Ext3) corruption (access beyond end)	2008-06
Ext3	Debian, #425534	linux-2.6: Ext3 filesystem corruption	2008-09
Ext3	Debian, #533616	linux-image-2.6.29-2-amd64: occasional Ext3 filesystem corruption	2009-06
Ext3	Redhat, #515529	ENOSPC during fsstress leads to filesystem corruption on ext2, Ext3, and Ext4	2010-03
Ext3	https://lkml.org/lkml/2011/6/16/99	Ext3: Fix fs corruption when make_indexed_dir() fails	2011-06
Ext3	Redhat, #658391	Data corruption: resume from hibernate always ends up with EXT3 fs errors	2012-03
Btrfs	https://lkml.org/lkml/2009/8/21/45	Btrfs rb corruption fix	2009-08
Btrfs	https://lkml.org/lkml/2010/2/25/376	[2.6.33 regression] Btrfs mount causes memory corruption	2010-02
Btrfs	https://lkml.org/lkml/2010/11/8/248	DM-CRYPT: Scale to multiple CPUs v3 on 2.6.37-rc* ?	2010-09
Btrfs	https://lkml.org/lkml/2011/2/9/172	[PATCH] Btrfs: prevent heap corruption in Btrfs_ioctl_space_info()	2011-02
Btrfs	https://lkml.org/lkml/2011/4/26/304	Btrfs updates (slab corruption in Btrfs fitrim support)	2011-04

file systems. This problem is getting worse because disk capacities are growing faster than disk bandwidth and seek time [37]. Furthermore, the consistency check is run after the fact, often after a system crash occurs or even less frequently with journaling file systems. Thus an error may propagate and cause significant damage, making repair a non-trivial process [33]. For example, Section 4.4 shows that a single byte corruption may cause repair to fail.

To minimize the need for offline recovery methods, our aim is to verify file-system metadata consistency at runtime. Metadata is more vulnerable to corruption by file system bugs because the file system directly manipulates the contents of metadata blocks. Metadata corruption may also result in significant loss of user data because a file system operating on incorrect metadata may overwrite existing data or render it inaccessible.

We present a system called Recon that aims to preserve metadata consistency in the face of *arbitrary* file-system bugs. Our approach leverages modern file systems that provide crash consistency using transactional methods, such as journaling [18, 78, 81] and shadow paging file systems [7, 38, 68]. Recon checks that each transaction being committed to disk preserves metadata consistency. We derive the checks, which we call consistency invariants, from the consistency rules used by the offline file system checker. A key challenge is to correctly interpret file system behaviour without relying on the file system code. Recon provides a block-layer framework for interpreting file system metadata and invariant checking.

An important benefit of Recon is its ability to convert fail-silent errors into detectable invariant violations, raising the possibility of combining Recon with file system recovery techniques such as Membrane [77], which are unable to handle silent failures.

Recon shows the feasibility of interpreting metadata and writing consistency invariants for the widely used Linux Ext3 and Btrfs file systems. Recon checks Ext3 invariants corresponding to most of the consistency properties checked by the e2fsck offline check program. It detects random and type-specific file-system corruption as effectively as e2fsck, with low memory and performance overhead. At the same time, our approach does not suffer from the limitations of offline checking described earlier because corruption is detected immediately. The rest of the chapter describes our approach in detail and presents the results of our initial evaluation.

4.1 Approach

The Recon system interposes between the file system and the storage device at the block layer and checks a set of consistency invariants before permitting metadata writes to reach the disk. We derive the invariants from the rules used by the file system checker. As an example, the e2fsck program checks that file system blocks are not doubly allocated. Our invariants check this property at runtime and thus prevent file-system bugs from causing any double allocation corruption on disk.

Figure 4.1 shows the architecture of the Recon system. Recon provides a framework for caching metadata blocks and an API for checking file-system specific invariants using its metadata cache. A separate cache is maintained because the file system cache is untrusted and because it allows checking the invariants efficiently (see Section 4.3 for details). Besides Ext3, we have also examined the consistency properties of the Linux Btrfs file system and implemented several Btrfs invariants. This chapter describes our initial experience with adapting our system for Btrfs; in Section 5.4 we provide some performance benchmarks of our Btrfs invariant checking along with the atomicity and durability invariant checking.

Our approach addresses three challenges: 1) *when* should the consistency properties be checked, 2) *what* properties should be checked, and 3) *how* should they be checked. In this section, we describe these challenges and how we address them. The caching framework and Recon APIs are described in Section 4.3.

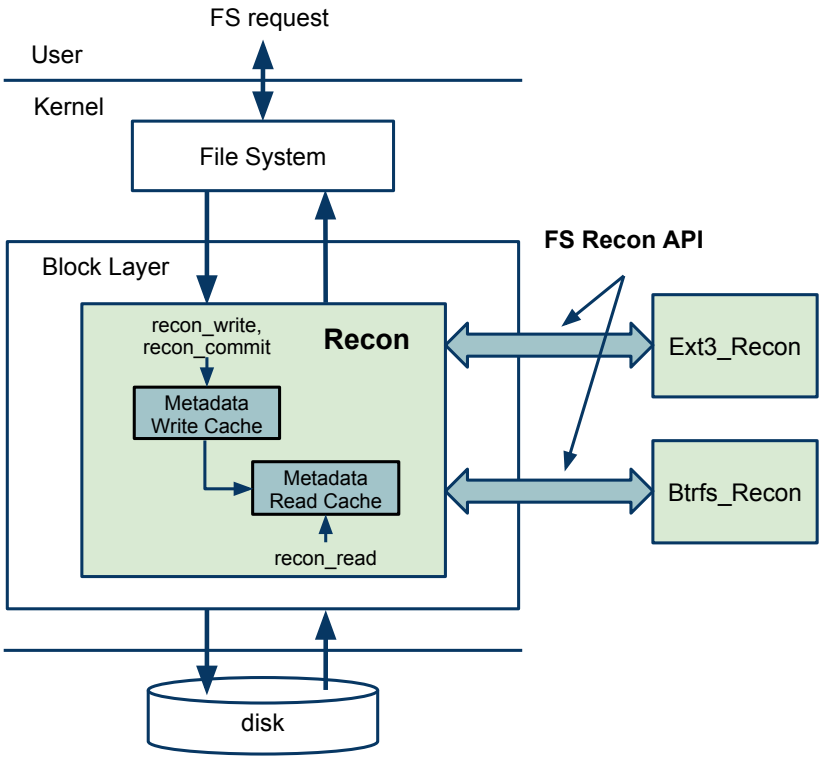


Figure 4.1: The Recon Architecture

4.1.1 When to Check Consistency?

The in-memory copies of metadata may be temporarily inconsistent during file system operation and so it is not easy to check consistency properties at arbitrary times. Instead, checks can be performed when the file system itself claims that metadata is consistent. For example, journaling and shadow-paging file systems are already designed to ensure crash consistency using transactional methods, wherein disk blocks from one or more operations, such as the creation of a directory and a file write, are grouped into transactions. Transaction commits are well-defined points at which the file system believes that it is consistent, and hence transaction boundaries serve as convenient vantage points for verifying consistency properties. Recon checks transactions *before* they commit, thereby ensuring that a committed transaction is consistent, even in the face of arbitrary file system bugs.[2]

Checking consistency for shadow paging systems is relatively straightforward because all transaction data is written to disk before the commit block. For example, Btrfs writes all blocks in a transaction, and then commits the transaction by writing its superblock. Recon checks each transaction before the superblock is written to disk.

Checking consistency for journaling file systems is more complicated because transaction data is written to disk both before and after the commit block. For example, Ext3 writes metadata to disk in several steps: 1) write metadata to journal, 2) write commit block to journal, at which point the transaction is committed, 3) write (or checkpoint) metadata to its final destination on disk, and 4) free space in the journal.

During Step 1, Recon copies metadata blocks into its write cache, giving it a view of all the updates in a transaction. Then it checks the Ext3 transaction in Step 2, i.e., before the commit block is written to the journal, which ensures that all blocks in the transaction are checked for consistency before they become durable. Checking

[2]Implementing consistency invariants for soft update file systems [27] that provide consistency after each write but do not use transactions should be possible but will likely be more complicated.

consistency after commit could lead to checkpointing a corrupt block, and furthermore it would not be possible to undo such corruption. Besides checking consistency at commit, we also need to verify the checkpointing process. This step requires checking that all the committed blocks and their contents are checkpointed correctly.

4.1.2 What Consistency Properties to Check?

Identifying the correct consistency properties is challenging because the behaviour of the file system is not formally specified. Fortunately, we can derive an informal specification of metadata consistency properties from offline file-system consistency checkers, such as the Linux e2fsck program. For example, Gunawi et al. found that the Linux e2fsck program checks 121 properties that are common to both ext2 and Ext3 file systems and some Ext3 journal properties and optional features [33].

These consistency properties define what it means to have consistent metadata on disk. Our aim is to ensure that any metadata committed to disk will maintain these same consistency properties. Unfortunately, consistency properties are *global* statements about the file system. For example, a simple check implemented by e2fsck is that the deletion times of *all* used inodes are zero. Determining the in-use status of all inodes, and checking the deletion time of all used inodes is infeasible at every transaction commit. Similarly, another consistency property is that *all* live data blocks are marked in the block bitmap. Checking these global properties requires a full disk scan.

Instead, we derive a *consistency invariant* from each consistency property. The invariant is a local assertion that must hold for a transaction to preserve the corresponding file system consistency property. For example, consider the "all live data blocks are marked in the block bitmap" property. The corresponding consistency invariant is that a transaction that makes a data block live (i.e., by adding a pointer to the block) must also contain a corresponding bit-flip (from 0 to 1) in the block bitmap within the same transaction, i.e., the invariant is "block pointer set from 0 to N $\Leftrightarrow$ bit N set in bitmap". This invariant can be checked by examining only the updated blocks, i.e., the updated pointer block and the updated block bitmap must be part of the same transaction. We describe this invariant in more detail in Section 4.2.2.

We structure a consistency invariant as an implication, $A \Rightarrow B$. The premise A *always* involves an update to some data structure field, and hence checking the invariant is triggered by a change in that field. When such an update occurs then the conclusion B must be true to preserve the invariant. If a converse $B \Rightarrow A$ invariant also exists, then we refer to the two invariants as a biconditional invariant $A \Leftrightarrow B$, as shown in the example above.

We rely on the ability to convert consistency properties requiring global information into invariants that can be checked using information "local" to the transaction, as described in the previous example. Such a transformation must be possible because file systems keep themselves consistent without examining the entire disk state. In other words, our invariant checking should not require much more data than the file system itself needs for its operations. Section 4.4 shows that this is indeed the case because Recon has low overheads.

Finally, our invariant checking approach relies on an inductive argument. It assumes that the file system is consistent before each transaction. If the updates in the transaction meet the consistency invariants, the file system will remain consistent after the transaction. Likewise, if an invariant is violated, there is potential for data loss or incorrect data being returned to applications. Section 4.1.4 provides more details about our assumptions.

4.1.3 How to Check Consistency Invariants?

Consistency invariants are expressed in terms of logical file-system data structures, such as current and updated values of block pointers, bits in block bitmap, etc. However, Recon needs to observe physical blocks below the

file system because it cannot trust a buggy file system to provide the correct logical data structure information. We bridge this semantic gap by inferring the types of metadata blocks when they are read or written, which allows parsing and interpreting them, similar to semantically smart disks [75]. Then Recon checks invariants on the typed blocks at commit points. Interpreting file system metadata and checking the invariants are the tasks of the file-system specific modules of Recon (Figure 4.1).

Metadata Interpretation

Block typing and metadata interpretation depend on the idea that file systems access metadata by following a graph of pointers. Ultimately, this graph is rooted at a known location or can otherwise be identified (e.g., by a magic number or a hash), which we call the "superblock". As long as Recon knows the structure of each type of metadata block, including the types of the blocks that it may point to, Recon can interpret all of the metadata that is read or written.

For example, a pointer to a block is read before the pointed-to block is read, which we call the pointer-before-block assumption. These pointers may be explicit block pointers or are implied by the structure of the file system. For example, Ext3 will read an inode containing a pointer to an indirect block before reading the indirect block. When an inode block is read, Recon copies it into its read cache, parses the inodes in the block, and then creates identification information for any metadata blocks pointed to by the inodes. For instance, if the EXT3_IND_BLOCK field in an inode has a non-zero value, X, Recon will associate two pieces of information with block number X: (1) the "indirect block" type, and (2) the inode number of the inode that points to it. As a result, Recon recognizes an indirect block and its containing inode when block X is read.

Similarly, the block group descriptor (BGD) tables in Ext3 describe the locations of inode blocks and inode and block allocation bitmaps. The BGD tables must be read before any of the blocks that they point to, allowing Recon to create block identification information for inode and bitmap blocks. This block identification is bootstrapped using the superblock, which exists at a known location. The block identification information is maintained in the Recon caches, as described in more detail in Section 4.3.3.

When a metadata block is newly allocated in a transaction, Recon does not know its type initially. In this case, there must exist an updated metadata block in the transaction with a known type that points to this unclassified block directly or indirectly, or else the newly allocated block would not be reachable in the file system. By following the path of pointers from the known metadata block to the newly allocated block, Recon can always create block identification information for newly allocated blocks.

For example, suppose a block is allocated to be an indirect block of a file. If the file already existed then its inode block must have been read and updated in the transaction. Since the inode block was read previously, Recon knows its type and can determine the type of the newly allocated indirect block. Similarly, if the file did not exist then its parent directory must have existed and been updated, which helps determine the types of the (possibly newly allocated) inode block and then the indirect block. Determining the types of newly allocated blocks may require multiple passes over the blocks updated in the transaction. At the end, all new metadata blocks associated with the transaction must be typed. Otherwise, an untyped block would not be accessible from the committed state of the file system, violating the pointer-before-block assumption.

Commit Processing

At commit, Recon uses the block identification information to determine the data structures in the most recent versions of each of the (updated) transaction blocks, available in the Recon write cache. These data structures

```
[Inode, 12, block[1], 0, 22717]   ; In inode 12, direct block ptr 1 is set to
                                  ; block 22717
[BBM, 22717, 0, 0, 1]             ; Block 22717 is marked allocated in block bitmap
[BGD, 0, free_blocks, 1500, 1499] ; In block group 0, number of free blocks
                                  ; decreases by 1
[Inode, 12, i_size, 4010, 7249]   ; i_size field increases from 4010 to 7249 bytes
[Inode, 12, i_blocks, 8, 16]      ; i_blocks is the number of sectors used by file
[Inode, 12, mtime, 1-18-12, 1-20-12] ; timestamp change
[Inode, 12, ctime, 1-16-12, 1-20-12] ; timestamp change
```

Figure 4.2: Change records when a block is added to a file

are compared with their previous versions, which are derived from the blocks in the Recon read cache, at the granularity of data structure fields. Each field update generates a logical *change record* with the format [*type, id, field, oldval, newval*].

The *type* specifies a data structure (e.g., inode, directory block, etc.). The *id* is the unique identifier of a specific object of the given type (e.g., inode number). The (type, id) pair allows locating the specific data structure in the file system image. The *field* is a field in the structure (e.g., inode size field) or a key from a set (e.g., directory entry name). The *oldval* and *newval* are the old and new values of the corresponding field. These records are generated for existing, newly allocated and deallocated metadata blocks. When an item is newly created or allocated, the oldval is ϕ (a sentinel value). Similarly, when an item is destroyed or deallocated, the newval is ϕ.

Figure 4.2 shows an example of a set of change records associated with an Ext3 transaction in which a single write operation increases the size of a file from one block to two blocks. Change records serve as an abstraction, cleanly separating the interpretation of physical metadata blocks from invariant checking on logical data structures. We show how invariants are implemented using change records in Section 4.2. When all invariants are checked successfully, the transaction is allowed to commit.

4.1.4 Fault Model

Our goal is to preserve file-system metadata consistency in the presence of arbitrary file-system bugs. We make three assumptions to provide this guarantee. First, we assume that the Recon code and its invariant checks are correct and immutable and the Recon metadata cache is protected. If these assumptions are incorrect, it is unlikely that an inconsistent transaction would pass our checks, because the file-system bug and our corrupted check would need to be correlated. Corrupting Recon may generate false alarms, indicating that a transaction is corrupt when it is actually consistent. This increases the likelihood of a fail-stop scenario, however, we believe that halting when a system is malfunctioning is preferable to silently corrupting data.

A hypervisor-based Recon implementation would provide stronger isolation of the Recon code and data from the kernel, helping ensure metadata consistency in the face of *arbitrary kernel* (instead of just file system) bugs.

Second, if the Ext3 file system writes a metadata block before Recon knows its type then Recon will assume that a data block is being written and will allow the operation. For example, a file system bug may corrupt the block number in a disk request structure and cause a misdirected write [4] to a metadata block. Recon will not detect this error because the write violates our pointer-before-block assumption, and Ext3 does not provide any other way to identify the block being updated.[3] In Chapter 5 we show how Ext3 can be augmented with a metadata

[3]We did not observe this problem because our fault injector corrupts metadata blocks but does not cause misdirected writes (see Section 4.4.2).

bitmap, making it feasible to detect this kind of corruption. Misdirected writes will not cause a problem with Btrfs because its extents are self-identifying, making it easier to distinguish between data and metadata block locations.

Finally, our inductive assumption about metadata consistency before each transaction (discussed in Section 4.1.2) requires correct functioning of the lower layers of the system, including the Linux block device layer and all hardware in the data path. It is possible to detect and recover from errors at these layers by using metadata checksums and redundancy. This functionality could be implemented at the block layer for the Ext3 file system [32]. The Btrfs file system already provides such functionality [68]. If these assumptions are not met, offline checking and repair should be used as a last resort.

4.2 Consistency Invariants

A file system checker verifies file system consistency by applying a comprehensive set of rules for detecting and optionally repairing inconsistencies. We are primarily interested in checking consistency properties and can reuse the rules associated with detecting, but not repairing, inconsistencies. We have applied our approach to the Ext3 and the Btrfs file systems. Below, we provide an overview of the consistency rules for these file systems.

The SQCK system [33] encapsulates the 121 checks of the Ext3 fsck program in a set of SQL queries. Although there is a close correspondence between SQCK queries and e2fsck checks, some SQCK queries combine multiple checks. Table 4.1 provides a breakdown of the number of rules checked by SQCK for different file-system data structures. We show 101 rules in Table 4.1, because the rest are used for repair. The simplest checks (lines starting with the word *Within*) examine individual structures (e.g., superblock fields, inode fields, and directory entries appear valid). Some checks ensure that pointers lie within an expected range. More complicated checks (lines starting with the word *Between*) ensure that block pointers (across all files) do not point to the same data blocks, and directories form a connected tree.

We have done a similar classification of the rules checked by the Btrfs checker, as shown in Table 4.2. Btrfs is an extent-based, B-tree file system that stores file-system metadata structures (e.g., inodes, directories, etc.) in B-tree leaves [68]. It uses a shadow-paging transaction model for updates and for supporting file-system snapshots. Extent allocation information is maintained in an extent B-tree, which serves the same purpose as Ext3 block bitmaps. The roots for all the B-trees are maintained in a top-level B-tree called the root tree. Although the Btrfs checker is still a work in progress (e.g., it performs no repair), currently it uses 30 rules for detecting inconsistencies. Of these, the first four rule sets (marked A-D in Table 4.2) are used to check the structure of the B-tree, while the rest deal with typical file-system objects such as inodes and directories.

Next, we provide several examples that show how we transform the consistency properties for various data structures shown in Tables 4.1 and 4.2 into invariants. An invariant is implemented by pattern matching change records. When such a match occurs, invariants may accumulate booking information, and then require some final processing at transaction commit.

4.2.1 Ext3 Immutable Fields, Range Checks

The Ext3 fsck program checks for valid values in several fields of the superblock and group descriptor table (rows A and B in Table 4.1). Many of these fields are initialized when a file system is created and should never be modified by a running file system. Invariants on these fields are implemented by pattern matching a change record of the form [Superblock, _, immutable_field, _, _], where immutable_field is the name of the field that should not change, and _ matches any value. The existence of this record indicates that the field was modified, and signals

Data Type		#rules
A	Within superblock	23
B	Within block group descriptors (BGD)	5
C	Within a single inode	28
D	Within a single directory	14
E	Between inode and directory entries	5
F	Between inode and its block pointers	2
G	Between inode, inode bitmap, orphan list	3
H	Between block bitmap and block pointers	5
I	Between block, inode bitmap, BGD table	3
J	Between directories	4
K	Bad blocks inode	7
L	Extended attributes ACL	2

Table 4.1: Number of Ext3/SQCK rules by datatype

a violation. Another similar class of consistency properties requires simple range checks on the values of given fields.

4.2.2 Ext3 Block Bitmap and Block Pointers

An important consistency property in Ext3 is that no data block may be doubly allocated, i.e., every block pointer (whether it is found in a live inode or indirect block) must be unique or 0. Checking this property would be expensive if we simply scanned all inodes and indirect blocks searching for another instance of the pointer.

The file system maintains this property without examining the entire disk state by using block allocation bitmaps (row H in Table 4.1), with the resulting consistency property being that "all live data blocks are marked in the block bitmap". The corresponding consistency invariant is that a transaction that makes a data block live (i.e., by adding a pointer to the block) must also contain a corresponding bit-flip (from 0 to 1) in the block bitmap within the same transaction, as shown below.

block pointer set to N from 0 $\Leftrightarrow$ bit N set in bitmap (1)

block pointer set to 0 from N $\Leftrightarrow$ bit N unset in bitmap (2)

These invariants involve relationships between different fields and require matching multiple change records. The left side of the first invariant is triggered by matching change records of the form [_, _ , block_pointer_field, 0, X], indicating a new pointer to block X. When such a match occurs, we insert a "new pointer" flag with key X into a rule-specific table. The right side of this (biconditional) invariant is triggered by matching [BBM, Y, _, 0, 1] records, indicating bit Y in the allocation bitmap is newly set. When this match occurs, we insert a "bit set" flag with key Y into the same table. During final processing, the implementation verifies that for each key in the table, both flags are set. Otherwise the invariant has been violated. For example, in the simple transaction shown in Figure 4.2, there is exactly one record matching each of the left and right sides of Invariant 1 shown above, and the values of X and Y are both 22717.

Invariants 1 and 2 ensure that when a block pointer is set, the corresponding bit in the bitmap is also set. However, we must also ensure that a pointer to the same block is set only once in a transaction, i.e., we must check for double allocation within a transaction. To do so, we simply count the number of times we see a block pointer set to a given block in the transaction:

Data Type		#rules
A	Within tree block	2
B	Between parent and child tree blocks	3
C	Between extent tree and extents	3
D	Within an extent item in extent tree	2
E	Between inodes and file system trees	2
F	Between inode and directory entries	4
G	Between inodes, inode refs and dir. entries	2
H	Within directory entries	1
I	Between inode, data extents, checksum tree	6
J	Between inode and orphan items	1
K	Between root tree and file system trees	3
L	Between root tree and orphan items	1

Table 4.2: Number of Btrfs rules by datatype

block pointer set to N ⇒

(count(block pointer==N) in transaction)==1 (3)

4.2.3 Ext3 Directories

The inter-directory consistency properties essentially ensure that the directory tree forms a single, bidirected[4] tree (row J in Table 4.1). This complex consistency property requires two biconditional and two regular invariants. Whenever a directory is linked (or its ".." entry changes), Invariant 4 checks that the directory's parent (child) has the directory as its child (parent). This check also ensures that a directory does not have multiple parents. When a directory is unlinked (or moved), Invariant 5 checks that it is unlinked on both sides (although not shown, we also check that an unlinked directory is empty). When a directory's "." entry is updated, Invariant 6 checks that the "." entry points to itself.

[Dir, C, "..", _, P] ⇔ [Dir, P, nm, _, C] and (nm != "..") (4)

[Dir, C, "..", P, _] ⇔ [Dir, P, nm, C, _] and (nm != "..") (5)

[Dir, D1, ".", _, D2] ⇒ D1==D2 (6)

[Dir, _, "..", _, P] ⇒ is_ancestor(ROOT, P) (7)

Finally, Invariant 7 checks that a directory update does not cause cycles. Invariants 4 and 5 do not prohibit cycles. For example, suppose that the file system allows the command "mv /a /a/b" to complete successfully. This update would be allowed by the Invariants 4 and 5, but it would create a disconnected cycle consisting of a and b. Invariant 7 checks for cycles when a directory's parent entry (the ".." entry) is updated. It ensures that the chain of parent directories eventually reaches the root directory, or a cycle is detected. The is_ancestor() primitive operates on the Recon metadata caches described in Section 4.3.

[4] A bidirected tree is the directed graph obtained from an undirected tree by replacing each edge by two directed edges in opposite directions.

4.2.4 Btrfs Inode and Directory Entries

Metadata structures in Btrfs are indexed by a 17-byte key consisting of the tuple (objectID, type, offset). ObjectID is roughly analogous to an inode number in Ext3. The type field determines the type of the structure, and the meaning of "offset" depends on the type. Each key is unique within a Btrfs tree, so the unique (type, id) pair for our change records consists of (type, (tree id, objectid, offset)).

A Btrfs consistency property is that the inode associated with a directory item (that is, a Btrfs directory entry) has a directory mode (row F in Table 4.2). An invariant derived from this property is that when we add a new directory item, there must exist an appropriate inode item after transaction commit. We can represent this as:

[DIR_ITEM, (T, I, _), _, ϕ, _] $\Rightarrow$
  exists(T, I, INODE_ITEM, 0) and
  ISDIR(get_item(T, I, INODE_ITEM, 0).mode)

The left hand side matches a directory item within snapshot tree T and objectid I that is being newly created. This invariant asserts that 1) there is a matching inode item, and 2) its mode is of directory type. The exists() primitive returns true if the given item can be found in tree T, and the get_item primitive obtains the contents of the item, allowing us to check the mode. These primitives operate on the Recon metadata caches.

4.3 Implementation

This section describes the implementation of the Recon system. Section 4.3.1 describes the structure of the Recon metadata caches. Next, Section 4.3.2 describes the main interfaces provided by the Recon system and how they are invoked during block IO operations. Section 4.3.3 describes how we identify blocks when they are read from disk. The file-system specific processing that is needed for metadata interpretation and invariant checking is presented in Section 4.3.4. This processing is performed on the blocks cached by Recon. Section 4.3.5 describes our method for evicting blocks from the Recon caches to limit memory consumption. Then, Section 4.3.6 describes our strategy for synchronizing block reads issued by the file system code and the Recon code. Finally, Section 4.3.7 describes several strategies for dealing with invariant violations.

4.3.1 Structure of Metadata Caches

The Recon Read and Write metadata caches hold copies of filesystem metadata blocks, along with additional information to identify and manage the blocks. All of the key Recon operations work with these caches, therefore it is useful to understand the basic structure of the cache and cache entries, before describing the Recon operations in detail.

We use standard Linux hash table structures to implement the two caches, with the on-disk location of a block (i.e., its sector number) serving as the hash key. The basic cache structure is illustrated in Figure 4.3. Each cache entry consists of a block header, which includes a pointer to a block buffer that stores the contents of the metadata block. The block buffers in the Recon caches are marked read-only after they are filled with a metadata block, to protect against errant memory writes. Recon itself never modifies metadata blocks, so the buffers remain read-only until the block is evicted from the cache.

The rest of the block header can be logically divided into three components: 1) hashing, 2) identity, and 3) bookkeeping information, as shown on the right side of Figure 4.3. The hashing information, namely a Linux

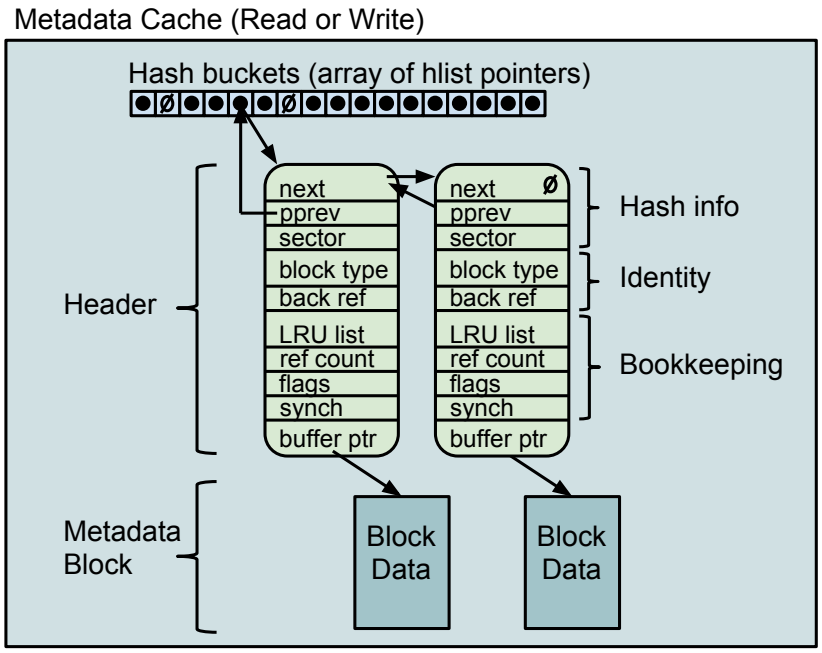


Figure 4.3: Structure of entries in Recon metadata caches

Generic API	
recon_init	initializes recon state
recon_read	caches block in metadata read cache
recon_write	caches block in metadata write cache
recon_commit	performs metadata interpretation and invariant checking

Table 4.3: Generic Recon API

`hlist_node` and the hash key, is needed to maintain the hash chain. The identity information helps identify the block, and it includes the block type (e.g., `indirect block` in Ext3) and a back reference to the metadata block that points to the current block (e.g., the containing inode number for an indirect block). Section 4.3.3 describes how Recon records this information. Note that for file systems like Btrfs, where metadata blocks are self-identifying, Recon does not need to keep this information in the block header. Finally, the header also stores bookkeeping information that is used by Recon to manage the blocks in the cache, including an LRU list and a reference count for cache replacement (see Section 4.3.5), various flags to record the block status, and a synchronization variable to coordinate block reads issued by the file system with those issued by Recon itself (see Section 4.3.6).

4.3.2 Invoking Recon Operations

We use the Linux device mapper framework to interpose on all file system I/O requests at the block layer, as shown in Figure 4.1. Recon was initially implemented using Linux 2.6.27, and later ported to 2.6.35 (for Btrfs support) and 3.8.11 (in Chapter 5). Our code in the mapper framework invokes the Recon API functions shown in Table 4.3. On a block read, the recon_read function caches a metadata block in the Recon read cache. The read cache allows accessing the pre-update file-system state (i.e., disk state) efficiently. Its contents are trusted because its blocks have been verified previously. On a block write, recon_write caches an updated metadata block in the

Recon write cache, which allows accessing the post-update file-system state. Note that the post-update state is derived by superposing the write cache on the read cache, similar to a copy-on-write system. The write cache may contain corrupt data and thus any code accessing this cache must perform careful validation. On a transaction commit, recon_commit performs metadata block interpretation and invariant checking, as described in more detail in Section 4.3.4 below.

The recon_write and recon_commit functions are supposed to be called on a block write and transaction commit, respectively. Our implementation can invoke these two functions in two different ways, from the transaction subsystem (*transaction-layer interception*) or from the block layer (*block-layer interception*). Both implementations ensure that 1) recon_write is invoked on all updated metadata blocks, and 2) recon_commit is called before the commit block reaches the disk. Block-layer interception provides stronger isolation guarantees, and is necessary for checking the integrity of the transaction subsystem, described in Chapter 5.

With block-layer interception, recon_write is invoked at block writes. There are two challenges with this approach. First, we need to separate metadata blocks from data blocks because Recon operates on metadata blocks and we do not want to cache data blocks on each write. Second, we need to detect commits because they are not directly visible at the block layer. For Btrfs, the block-layer implementation can easily distinguish metadata writes because they are directed to designated regions on disk called Btrfs chunks. Btrfs commits occur when the superblock is written, which is easy to detect because the superblock is in a known location.

The situation is more complicated for Ext3, because it mixes dynamically allocated metadata blocks with data blocks in the same regions on disk. This makes it hard to distinguish metadata from data blocks at the time they are written; it is only at commit time, when we interpret the new pointers that are part of the transaction, that we can identify the newly allocated metadata blocks. Fortunately, for Ext3 the default mode is metadata-only journaling, which provides a way to separate data from metadata blocks based on the destination of the write. Since only metadata blocks are written to the journal, we can ignore blocks that are not written to the journal area. Hence, we invoke the recon_write function on each journal block write, which copies the metadata block into the write cache, indexed by its location in the journal.

Commits in Ext3 can be detected by interpreting the content of blocks written to the journal, as part of the recon_write operation. Interpreting journal writes also helps to determine the final locations of the updated metadata blocks, since this information must be stored in the journal itself (e.g., in JBD's descriptor blocks). When the commit block is identified, we reindex the updated metadata blocks in the write cache from their journal locations to their final locations. We also drop any journal metadata blocks (e.g., journal descriptor blocks) from the write cache since they will not be checkpointed. We then invoke recon_commit to perform invariant checking before allowing the commit block to be written to the disk. While this implementation is more complicated, it removes any dependency on the journaling code.

With transaction-layer interception, recon_write is not invoked on a block write directly. Instead, the file system's transaction commit code is modified to invoke recon_write on the updated metadata blocks (stored in the transaction layer) and then to invoke recon_commit. Both functions are called just before writing the commit block. This method is simpler to implement, but it makes us dependent on the transaction layer code, such as JBD in Ext3, or the transaction commit code in Btrfs. Also, it does not allow us to verify the Ext3 checkpointing code since we intercept commits but not each block (journal or checkpointing) write. Similarly, for Btrfs, it does not allow checking for Btrfs copy-on-write semantics, i.e., already allocated metadata blocks should not be overwritten by a block write.

We initially implemented transaction-layer interception for the Ext3 file system. After gaining more experience with Ext3 and our Recon implementation for Ext3, we implemented block-layer interception for the Ext3

FS Recon API	Invoked by	
process_read	recon_read	determines whether a metadata block is accessed, provides type and id information for pointed-to metadata blocks
process_write	recon_write	determines whether a metadata block is written
process_block	recon_commit	provides type and id information for newly allocated metadata blocks
process_txn	recon_commit	generates change records
check_txn	recon_commit	checks invariants using change records and metadata read/write caches

Table 4.4: File-system specific Recon API

file system. In this paper, we evaluate the Ext3 block-layer implementation because it cleanly separates the Ext3 and the Recon implementations, providing stronger fault isolation.

4.3.3 Block Identification

Each block in the two Recon caches is associated with a block header that helps identify the block, as shown in Figure 4.3. The block header maintains the location of the block on disk, and block-specific identification information such as the block type (e.g., Ext3 indirect block) and other information needed to unambiguously identify the block when it is read from disk (e.g., the inode number that points to the indirect block). As described in Section 4.1.3, the block identification information is used by the file-system specific code to identify and interpret the contents of metadata blocks. For example, when an inode block is read by the Ext3 file system, Recon parses the inode block and creates a block header for all blocks pointed to by the inodes in the inode block. At this point, these block headers do not have a block buffer (i.e., the block contents) associated with them. When a pointed-to block is read, Recon looks up the read cache to find a block header for the block. As a result, it knows the type of the block and the containing inode, and can thus parse the pointed-to block. When a block is read, its header must exist or else the pointer-before-block assumption would be invalid. After the block is read, the block contents are copied to a buffer and this block buffer is attached to the block header.

4.3.4 File-System Specific Processing

The Recon framework invokes file-system specific API functions for metadata interpretation and invariant checking, as shown in Table 4.4. Below, we describe these functions in more detail.

Recon_read and Recon_write

The process_read function is invoked by recon_read to parse a metadata block and create block identification information, i.e., block headers, for pointed-to blocks. This function is also used to distinguish between data and metadata on the read path so that only metadata blocks are cached in the Recon read cache. The process_write function is invoked by recon_write in the block-layer implementation to distinguish between data and metadata on the write path, as explained in Section 4.3.2 earlier. It is not needed in the transaction-layer implementation because the transaction layer provides the set of updated metadata blocks.

Recon_commit

When a transaction commits, recon_commit checks invariants by invoking the rest of the file-system specific calls shown in Table 4.4. The process_block function is similar to the process_read function and parses updated metadata blocks. It is invoked on all the blocks in the write cache (i.e., each updated or newly allocated metadata block). This function must validate the updated blocks by checking that any pointers, strings and size fields within the block have reasonable values so that further processing is not compromised. Recon ignores blocks of an unknown type (unknown blocks) and only processes updated blocks whose types are known (known blocks). When an unknown block becomes known (because it is pointed to by a known block, as explained in Section 4.1.3), it is added to the queue of blocks being processed. This processing works in multiple passes over the queue of blocks and stops when a pass does not convert any unknown blocks to known blocks.

At this point, if any unknown blocks remain, they will not be accessible from the committed state of the file system. In the case of Ext3, Recon signals a reachability invariant violation because the journal blocks are written in transaction sequence order and hence this case should never arise during normal operation. In Btrfs, we found that such unknown blocks arise frequently because blocks from the next transaction can be written to disk before the current transaction commits, and as expected, the next transaction blocks are unreachable from the current transaction blocks. As a result, the Btrfs implementation delays processing unknown blocks until the next transaction.

The process_block function helps determine the type of blocks (i.e., known blocks) and the data structures within known blocks. Next, the process_txn function compares updated data structures within known blocks with their previous versions (available from the read cache) to derive a set of change records. The previous version of a data structure is uniquely determined by the (type, id) pair of the change record. In Ext3, the type is determined by block type, and the id is typically an inode number or a block number. In Btrfs, the type and id are determined by the tree and the key, as discussed in Section 4.2.4.

While the process of comparing data structures is clearly file-system specific, we found two common cases. When data structures have fixed size, such as inodes in Ext3 and most items in Btrfs, we use a simple byte-level diff that is driven by tables that describe the layout of the data structures. These tables are generated from the data structures using C macros. When data structures themselves contain sets of variable-sized smaller items, such as directory entries in Ext3, or extent items in Btrfs, we use a set-intersection method to derive three sets consisting of new items, deleted items and modified items. Change records can be generated from these sets, using the identity of the containing item (e.g., directory inode) and some key as field name (such as the "name" for directory entries). The check_txn function implements invariant checking as described using examples in Section 4.2.

When all applicable invariants have been checked successfully, the transaction is allowed to commit. After that the known blocks in the write cache are moved to the read cache (and the previous read cache versions of those blocks are destroyed), thus updating Recon's view of file-system state.

4.3.5 Cache Pinning and Eviction

We control the amount of memory used by the Recon caches with a simple LRU mechanism for replacing blocks from the read cache when it grows beyond a user-configurable limit. All read cache blocks are pinned during the recon_commit processing described in Section 4.3.4 to simplify implementation. We expect that recon_commit will run quickly because the blocks needed for commit processing have likely been read by the file system recently and so they will not need to be read from disk to populate the read cache.

We pin the Recon write cache for the duration of the transaction because we will need these blocks for checking invariants. This approach is similar to the Ext3 file system pinning its journal blocks for performance. However, we could unpin a block once it reaches disk, e.g., the journal in Ext3.

In addition, Recon allows file-system specific code to pin read cache blocks at any time. We describe a use of such pinning in Section 4.3.5 below.

Similar to a file-system buffer cache, the Recon caches do not need to persist across reboots, for two reasons: 1) all committed transactions have already been verified, and 2) the recovery code for checkpointing committed transactions in Ext3 can be verified independently, without needing the pre-crash contents of the Recon caches.

Pinning Journal Blocks in Ext3 File System

Recall from Section 4.3.4 that the known blocks in the write cache are moved to the read cache after commit. At this point, we can unpin the read cache because all the blocks in the cache are on disk (e.g., either in the journal or the checkpointed location in Ext3). However, we avoid evicting blocks that are in the journal because our transaction-layer Ext3 implementation does not track the location of blocks in the journal. Suppose that a block has just been moved from the write cache to the read cache and is currently in the journal but has not yet been checkpointed. Say that the cache eviction policy then evicts this block, and then Recon re-reads this block in its read cache. It may need to do so, for example, if the next transaction wrote to this block without reading its contents from disk and Recon needs to compare the updated block with its previous version. In this case, the transaction-layer Ext3 implementation would read a stale version of the block from the final location rather than the previous version in the journal because this implementation does not track the location of blocks in the journal. It may seem that the solution is to pin the read cache blocks that are in the journal blocks but again this implementation doesn't know about the journal blocks. Instead, we keep a list of most recently updated blocks in the read cache. This list contains as many blocks as it takes to fill the journal and we pin these blocks. Once a block is evicted from this list, it must have been checkpointed, or else it would have been overwritten in the journal, and so we can unpin it.

The block-layer Ext3 implementation does track the location of blocks in the journal and hence could unpin blocks as they are moved to the read cache, but currently, we still use the recently updated blocks strategy to pin the read cache blocks that may be in the journal.

Evicting Block Headers

We can evict blocks from the Recon read cache as described above, but we currently do not evict block headers. Recall that headers are created for pointed-to blocks when the block that points to them is read and parsed by the file-system specific process_read function. These headers contain critical block identification information (location on disk, block type and a back reference to the containing block) that allows the block to be parsed when it is later read from disk. If we evicted the header along with the block data, we would no longer be able to identify the block type (or its containing block) when it was re-read from the disk. For example, consider a block *A* that points to block *B*. When block *A* is read, Recon identifies the pointer to block *B* and creates a header for block *B*. Now suppose that the file system also reads *B*, causing Recon to attach a copy of block *B* to the header in the read cache, and that both the block data and the block header for *B* are evicted at some point. On a subsequent read of block *B*, Recon would not know its type, and would not even identify it as metadata. The reason is that the pointer-before-block assumption only holds when a block is read for the first time. The benefit of not evicting headers is that once a block is attached to its header, it does not need to be reparsed again, even if the block (but

not its header) is evicted from the Recon cache. For example, if block *A* was evicted from the Recon cache and later re-read again, then it would not need to be parsed since the header for block *B* still exists in the Recon cache. In our current implementation, headers are freed only when the block that they describe ceases to be metadata (e.g., because it has been freed by the file system in a transaction).

The obvious drawback of the current implementation is that the headers consume memory for every metadata block that is known to Recon, and may exhaust available memory. Our Ext3 implementation on a 64-bit system (where pointers are 8 bytes) uses a 68 byte header for each 4K metadata block, which can add up to a substantial memory overhead for large file systems. As shown in Figure 4.3, part of the header consists of bookkeeping information, which is only relevant when the block data is attached to the header. We could restructure the headers so that this bookkeeping information could also be freed when the block data is evicted, leaving only the hash information and the critical block identification fields in memory, for a savings of 28 bytes per header. Even with this optimization, which we have not implemented, the memory consumption of headers needs further attention.

This memory overhead can be reduced in three ways. First, we could persist the headers to disk separate from the file system. However, this method adds disk access overhead and the headers would have to be written atomically with file system commits, using a method such as chained transactions [32]. Second, we could synchronize with the file-system buffer cache's eviction policy. A block header can only be evicted if we know that the pointer-before-block assumption must hold. That is, the header for block *B* can only be evicted if the file system buffer cache has already evicted block *B* **and** the block that points to *B*. We cannot evict the header for a block that the file system still holds in its cache, since the file system could update that block and Recon would not be able to verify the updates against the previous version. Similarly, we cannot evict the header for a block when the file system still holds the containing block in its cache. In the example above, we cannot evict the header for block *B* while the file system holds block *A* in its cache, since the file system could then re-read *B* and we would not be able to identify it. If, however, both *A* and *B* have been evicted from the file system cache, then we can evict the header for *B*. Based on the pointer-before-block assumption, the file system must re-read block *A* if it needs to read block *B* from disk. When block *A* is re-read, we would recreate the block *B* header information. The main issues with this approach are (i) it requires the buffer cache to inform the block layer when it is evicting blocks, (ii) it relies on the correctness of the buffer cache code, and (iii) there is extra complexity in keeping track of when it is safe to evict a header.

Finally, a third approach is to create a block header when reading the block rather than when reading its containing block (e.g., create a header for block *B* when it is read, rather than when block *A* is read). This approach requires that blocks store their type information (self-typing) and have a reference to their containing block (e.g., a back reference). When both these requirements are met, we say that the block is self-identifying. For example, the Ext3 file system would be self-identifying if all of its dynamically allocated metadata blocks such as directory and indirect blocks were tagged with an inode number, specifying the inode that contains them, and the offset in the inode where the block is located. With this information, we could evict a block header at any time and recreate it when the block is read next time. Unlike Ext3, the Btrfs file system is self-identifying. Each Btrfs metadata block has a header that stores the type (node or leaf) and the physical location of the block. Btrfs uses back references extensively, e.g., a child node has a back reference to its parent, allocated data and metadata extents have back references to every block containing a pointer to the extent, and inodes have back references to each of the directory entries that reference them (i.e., hard links). A Btrfs implementation could use this information to create a header for a block as it is read, and evict the header when the block is evicted.

4.3.6 Cache Synchronization

The recon caches are accessed by multiple threads that issue read or write requests from the file system layer in Linux. We use a simple synchronization strategy and protect both the read and write caches with a single lock variable. Each of the Recon API functions (recon_read, recon_write and recon_commit) acquires this lock at the beginning and releases it before returning. One complication with this locking strategy is that the recon_read function needs to be invoked when a read request completes and the data is available to be copied into the read cache. In Linux, "endio" callback functions can be registered to execute at I/O completion. By replacing the original "endio" callback with our own function, Recon can perform its processing when the read completes. However, these "endio" callbacks execute in interrupt context and are not allowed to block or sleep. Thus we cannot simply call recon_read (and potentially block while acquiring the lock) in this context. Instead, we put all the information about the I/O request into a work item and queue it up for processing in another thread – which we refer to as the *recon thread*. The recon thread pulls work off its queue, processes the request (calling recon_read in this case), and then finishes by invoking the "endio" callback function attached to the original read request, thus notifying the original requester that the read is now complete.

When a write request is submitted, we invoke recon_write directly from the thread making the write request so that modified blocks are processed before they are sent to disk. Recon_write adds the block to the write cache and checks to see if it is the commit block. If the result of recon_write indicates that the write is not a commit block, then the write request is submitted as usual upon return from recon_write. However, if it is the commit block, then we add a work item to the queue for the recon thread indicating that the transaction is complete and can be checked. The requesting thread then blocks waiting for notification from the recon thread that the commit block has been handled. This approach allows us to verify the consistency of the transaction before the commit block is written out to disk. The recon thread handles the request by calling recon_commit, and then signals the original thread when it is done checking the transaction. Upon receiving that signal, the write request is submitted as normal.

The recon thread takes work items off its work queue and calls recon_read or recon_commit as needed. In either case, the processing may require Recon to access a block that has been evicted from the read cache, causing it to do a synchronous read of the missing data. In our initial implementation, the recon thread held the cache lock while performing such reads. We now describe three optimizations to this basic scheme that we have implemented.

First, we release the cache lock while the recon thread is re-reading an evicted metadata block into the read cache. All modifications to the read cache (either due to recon_read or recon_commit) are performed by the recon thread itself, and so the cache cannot change while the recon thread is waiting for its I/O to complete. Releasing the lock during the I/O allows recon_write to operate on the write cache.

Second, we leverage the recon read cache to try to satisfy the file system's metadata read requests, instead of fetching them from disk. To do so, we interpose on read requests at the block layer before they are sent to disk. For each read request, we first (i) acquire the recon cache lock, (ii) search the read cache for the requested block, (iii) if found, copy the data from the read cache into the buffer supplied for the read request, and (iv) release the recon cache lock. This simple locking scheme ensures that the block cannot be evicted by the recon thread between the lookup and the data copy operation. If a read can be served from the recon read cache, we simply mark the I/O as complete after copying the data and return without sending the request to the disk.

Finally, we ensure that there are no duplicate reads issued for the same metadata block. Since the file system buffer cache and the Recon caches are independent, it is possible for the file system to issue a read request for a metadata block while the recon thread already has a request outstanding for the same block, and vice versa. Rather than reading the same block twice, we make the second request wait for the first to complete. We use some

additional bookkeeping state in the block header to synchronize the requests. When the recon thread needs to read a missing metadata block into the read cache, it first checks to see if a read for the block is already in progress. If so, it releases the cache lock and waits for notification that the read has completed. When it receives a wakeup, the recon thread reacquires the cache lock and copies the data from the original I/O buffer into the read cache. If the block is not already being read, then the recon thread marks the header to indicate that a read is in progress, releases the cache lock, and issues the read request itself. When the request completes, the recon thread checks to see if any file system thread is waiting for the same block, and if so, issues a wakeup to the waiting thread. The situation is similar for requests from the file system. Read requests first check the Recon read cache to see if the requested block can be copied from there. At the same time, we also check if the requested block is currently being read, and if so, the reading thread waits for notification from the recon thread that the block is now present in the cache. In this case, the block can be copied from the read cache when the thread receives this notification. If the metadata block is not in the read cache, and is not already being read, then we mark the header to indicate that the block is being read by a file system request (thereby forcing the recon thread to wait if it wants to read the same block). When the file system read request completes, we issue a wakeup to the recon thread, if it is waiting for that block. The actual synchronization strategy is made complicated by the fact that we cannot use a blocking lock to issue a wakeup notification in the endio function associated with the file system's read request, and we cannot simply hand off the notification work to the recon thread, since the recon thread is the one that may be sleeping, waiting for that notification. Instead we use an atomic variable in the block header to track when wakeups are needed. This can be checked with atomic operations in the endio function, thus avoiding blocking in interrupt context.

4.3.7 Handling Invariant Violation

The final problem for an online consistency checker like Recon is dealing with invariant violations. It is important to ensure that recovery from a violation is correct and so the safest strategy is to disable all further modifications to the file system to avoid corruption. The file system can then be unmounted and restarted manually or transparently to applications [77]. In this case, the file system is not corrupt but may have lost some data. If the ability to create a snapshot (e.g., a Btrfs snapshot) is available, then a snapshot could be created immediately, the problem reported, and then we could continue running the file system. It is important to isolate the snapshot from the buggy file system, e.g., by directing all further writes to a separate partition. In this case, data is preserved but the file system may be corrupt. Finally, it may be possible to repair file system data structures dynamically [20].

4.4 Evaluation

In this section, we evaluate our Recon implementation for Ext3 in terms of its 1) complexity, 2) ability to detect metadata corruption at runtime, and 3) its performance impact.

4.4.1 Completeness and Complexity

We have implemented all of the checks performed by the e2fsck file system checker, as encapsulated by the SQCK rules, for the mandatory file system features. Overall, we need only 31 invariants (vs 101 SQCK rules) because some properties are easier to verify at runtime. For example, a large number of fields in the superblock and block group descriptors are protected with the simple invariant that they should not be changed by a running

file system. We also avoid explicit range check invariants in several cases because they are naturally embedded in other invariants that must check for setting or clearing of bits in bitmaps. There are a small number of properties on optional features that we do not check, such as OS-specific fields in inodes and the extended attributes ACLs.

Our entire system consists of 4.1k lines of C code (kLOC), as measured by the cloc [19] tool. Of these, 1.5 kLOC are in the generic framework which can be reused across file systems, 1.8 kLOC are for interpreting the Ext3 metadata, and only 0.8 kLOC are involved in checking the invariants. The code to implement block-layer interception of metadata block writes and journal commit blocks requires just over 100 lines of code to interpret the journal structures (this is included in the 1.8 kLOC to interpret Ext3 metadata). With transaction-layer interception, we do not need to interpret the journal contents, but we become dependent on the journal checkpointing code, which adds another 311 lines.

The code required to do consistency checking is simpler than the file system code for several reasons. First, the implementation of each invariant check is independent of the other checks because each rule uses its own data structures to keep track of properties that must be verified. Second, the implementation of each rule is usually quite simple, requiring several lines of C to accumulate the necessary data and a few more (often just a single boolean expression) to verify. Finally, we have a much simpler concurrency model than the file system, because the buffers to be processed are fully controlled by Recon and protected by a single lock.

4.4.2 Ability to Detect Corruption

Evaluating resiliency against metadata corruption is tricky. To best represent real-world corruption scenarios, we would either inject subtle bugs in the file-system or reproduce known bugs. However, subtle bugs (i.e., bugs not easily found in a heavily-used file system) are hard to design or reproduce. Reproducing known bugs is difficult as they often depend on specific kernel versions, combinations of loadable modules, concurrency levels, or workloads. Instead, we settled for deliberately injecting corruption of bytes within metadata blocks. This mimics the corruption that could result from several types of bugs (e.g., setting values in arbitrary fields incorrectly) both within the file system or in the overall kernel. We injected both type-specific corruption, where we target specific metadata block types and fields, and fully random corruption where we corrupt a sequence of 1 to 8 bytes within some number of blocks in a transaction.

Setup We compare Recon against e2fsck by corrupting metadata just before it is committed to the journal. We begin each corruption experiment by creating and populating a fresh file system, to ensure that there are no errors initially. Next, we start a process that creates a background of I/O operations (specifically we run a kernel compile and clean, repeatedly). The corruptor then sleeps for 20-90 seconds, wakes up, and performs the requested corruption (type-specific or random). We record the corruption performed and whether or not Recon detected it. Next, we allow the transaction to commit, and then immediately prevent any future writes. This step ensures that the corruption is limited to the bytes that we selected, rather than the result of the file system acting further on corrupt data. Next, we unmount the file system, run e2fsck on it, and record whether it found and repaired any errors. Finally, we run e2fsck a second time to see if the file system is clean after the repairs, and then reboot the system for the next experiment. For these experiments, we use a 4 GB file mounted as a loop device for our file system. This simplified the restoration of the file system following each corruption experiment.

Our corruption framework can only corrupt blocks that the file system is already modifying in some transaction. In particular, we never corrupt the superblock since the running file system never includes writes to it. We do not consider this to be a serious limitation to our test results since nearly all superblock corruptions would be

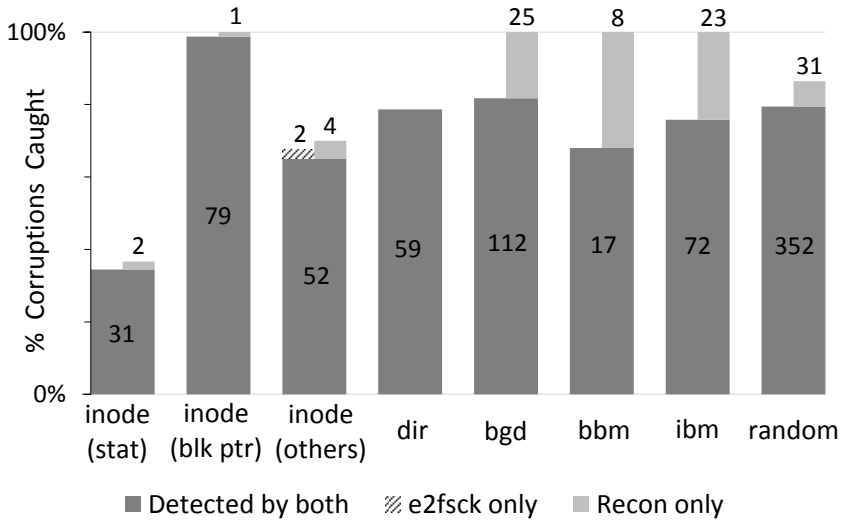


Figure 4.4: Comparison of corruption detection accuracy

trivially detected by Recon. Specifically, Recon protects most fields in the superblock with the invariant that they should not be modified at all, which is very easy to check.

Results Figure 4.4 summarizes the results of our corruption experiments. We show a wide bar and two stacked bars for each type of metadata corruption and random corruption. The wide bar shows the percent of corruptions (Y axis) that were caught by both e2fsck and Recon. The stacked bars show the percent of corruptions that were detected by only one checker. Numbers in the bars show the absolute number of corruptions detected.

For inodes, we present 3 sets of bars, representing different types of inode fields. The first group includes fields that are reported by "stat", the second group consists of all the block pointer fields, and the third group consists of everything else. Our coverage is nearly identical to e2fsck in all cases. Many of the inode stat fields are unrelated to file system consistency (e.g., the timestamps and userids) and are permitted to change arbitrarily, making it hard to detect corruption with either checker. However, both checkers are effective at catching corruption of block pointers. Recon achieves 100% in this case because it checks all inodes in a block being written to disk while e2fsck ignores unused inodes. Although file system consistency is not affected by changes to unused inodes, it is still useful to detect this corruption because it indicates a bug in the system. For the final set of inode fields, e2fsck detects an invalid flag setting that Recon does not check in two runs, while Recon catches corruption of some unused inode flags and a corruption of the dir_acl field that appears valid when checked by e2fsck after the fact in four runs.

For directory entries (dir), both checkers detect the same corruptions, with neither checker detecting corruption of the name field. For the other metadata types, Recon is more effective than e2fsck at detecting corruption, largely because it is able to take other runtime behaviour into account. For example, Recon achieves 100% detection for block group descriptor (bgd) corruption because most of these fields should not be changed by a running file system. Once corruption has reached the disk however, it is not always possible to distinguish the correct values from corrupted, but still valid, values. Similarly, Recon detects 100% of the block and inode bitmap (bbm and ibm, respectively) corruptions while e2fsck has a lower detection rate because it does not check unused parts of metadata blocks. For example, e2fsck does not check bits in the inode bitmap for non-existent inodes, or bits in the block bitmap for uninitialized block group descriptor table blocks. Recon's higher coverage on specific

Personality	Settings	Data Size
Webserver	nfiles=250k	3.9 GB
Webproxy	nfiles=500k	7.8 GB
Varmail	nfiles=250k	3.9 GB
Fileserver	nfiles=500k, filesize=32k	15.6 GB
MS-Networkfs	based on [58]	19.9 GB

Table 4.5: Benchmark Characteristics

metadata fields leads to higher coverage for fully random corruption as well. We expect that adding the final set of Ext3 invariants for OS-specific inode fields and extended attributes will help us detect all Ext3 structural consistency violations. However, neither checker can achieve 100% accuracy because some of the corruptions hit fields unrelated to structural consistency.

After e2fsck performs repair, it still detects errors in 28 out of 731 cases (3.8%), when it is run a second time on the "repaired" file system. Two of these failures occurred after a single byte was corrupted in a single metadata block. In our experiments, we unmount the file system and check it with e2fsck immediately after the corrupted transaction is committed to the journal. In reality, it is likely that the file system would continue operation with bad data for some time, making the chances of successful repair even lower. In these cases, Recon's ability to prevent corruption from reaching the on-disk metadata is particularly valuable.

4.4.3 Performance

Setup

All performance tests were done on a 1 TB Ext3-formatted file system on a dedicated Western Digital Caviar Black SATA drive. The host machine had 2 GB total RAM and dual 3 GHz Xeon CPUs. These experiments were performed using Linux kernel 2.6.27. We used the Linux port of Filebench (version 1.4.9.1) with the application emulation workload personalities[5]. We included the Networkfs personality, which supports a more sophisticated file system model, with a custom profile configured to match the metadata characteristics from a recent study of Windows desktops [58]. For Fileserver, we reduced the default file size to 32k to increase the metadata to data ratio in the file system. In all other cases, we used default parameter settings. Table 4.5 summarizes the basic characteristics of our benchmarks. The metadata load varies widely across the benchmarks, spanning the range of Recon cache sizes, causing misses in the cache. In particular, the Fileserver benchmark uses over 25k directories. The metadata consumed by directory entry blocks alone is greater than 100MB. The inodes for the directories and files would consume approximately 70MB if they were stored compactly, but Ext3 distributes allocation across different block groups, so unused inodes add to the metadata overhead. While the Networkfs benchmark involves more file data, the total number of files is lower because of the larger file size distribution.

Filebench emulates workloads using a variety of random variables to create an initial file system tree, to select files within the file system tree during the benchmark execution, and to select the operations (read, write, create, etc.) performed by each benchmark thread. As a result, each run of Filebench varies slightly within the parameters of the workload personality specified. To reduce this variation, we decided to ensure that each run starts with an identical file system tree. Since we are concerned with the performance of metadata operations, we especially wanted to ensure that each benchmark run started with the identical layout of the file system metadata on disk. To do so, we ran the create phase of the benchmark once for each workload and for each file system journal size to generate the initial file system tree for each test case. We then recorded the content and location of all the

[5]The OLTP personality did not work in the version we obtained.

metadata blocks in the file system for later reuse. During benchmarking, we begin with a reboot of the machine, and a restore of the initial file system metadata blocks to the raw disk device. Restoring the metadata blocks to the same locations on disk is sufficient to recreate the file system tree, since Filebench does not care about the data in the files. We set the Filebench profiles to reuse the existing file system tree, rather than creating a fresh one. To support file system reuse, we had to make a minor change to the way Filebench populates the initial tree, to guarantee that it would select the same initial set of files each time.

The benchmarks are run for one hour for all workloads to ensure that we capture steady-state behaviour with Recon. We report the performance of Recon compared to native Ext3 for the five Filebench workloads (Figure 4.5).

Our current implementation (described in Section 4.3) cannot evict blocks from our metadata cache that have not yet been checkpointed to the file system. Thus, the metadata cache size must be larger than the journal size. However, any memory consumed by Recon's metadata cache reduces the memory available for the file system cache by the same amount because Linux implements a shared page cache. We present results for three different cache/journal sizes, for both native and Recon performance. Although we begin with identical initial file system trees, Filebench still uses a variety of random variables for file and operation selection. Thus, there is still some natural performance variation across runs. Since this is representative of behaviour "in the wild", we report the average of 6 runs with error bars. All tests are done with cold caches on a freshly booted system.

Results

Figure 4.5 presents the overall Filebench throughput (reported as operations per second) for native Ext3 and Ext3 with Recon checking metadata consistency at the block layer. We see that the impact of Recon varies with the workload profile and the metadata cache size. For the read-intensive workloads (webserver and webproxy) the Filebench throughput is almost unchanged when Recon is enabled. With our smallest metadata cache size (64MB), we see only a 3.4% drop in throughput for webserver. The remaining workloads perform more metadata updates, and therefore Recon's checking of metadata consistency has a larger impact, particularly for small cache sizes. The fileserver and ms_nfs workloads show similar trends, with a worst-case overhead of 18% for fileserver and 21% for ms_nfs with our smallest cache size. As the cache size increases, the overhead is generally reduced. The one exception to this trend is the varmail workload personality where performance degrades slightly at the largest Recon cache size. We believe this is the result of memory pressure, as our increased metadata cache size decreases the amount of memory available to the file system buffer cache. Overall, a 128MB metadata cache with a 64MB journal results in reasonable overheads for all workloads. Further increases in the metadata cache size can further reduce the overhead for most workloads. Given the growth in main memory sizes, these are quite modest memory requirements for the reliability benefits that Recon can deliver.

To better understand the impact of Recon on file system workloads, we recorded a number of statistics within the Recon system itself. Recon can affect the measured throughput of workloads in several ways. First, Recon processing will add extra delay to file system read, write and commit operations. Since writes are usually performed asynchronously, and the recon_write processing is small, we do not expect the extra write delay imposed by Recon to have a significant impact. Extra delay on read operations, however, will have an impact on the workloads' throughput, since reading threads are usually blocked until the read completes. Although the recon_read processing is also small, the read delay can be significant because all read processing is performed by a single recon thread, and the same thread is responsible for checking consistency at transaction commit time. The effect of extra delay added to transaction commits depends on whether the foreground workload is waiting for the commit to complete or not. In workloads that sync frequently, such as varmail, the time to commit a transaction has a

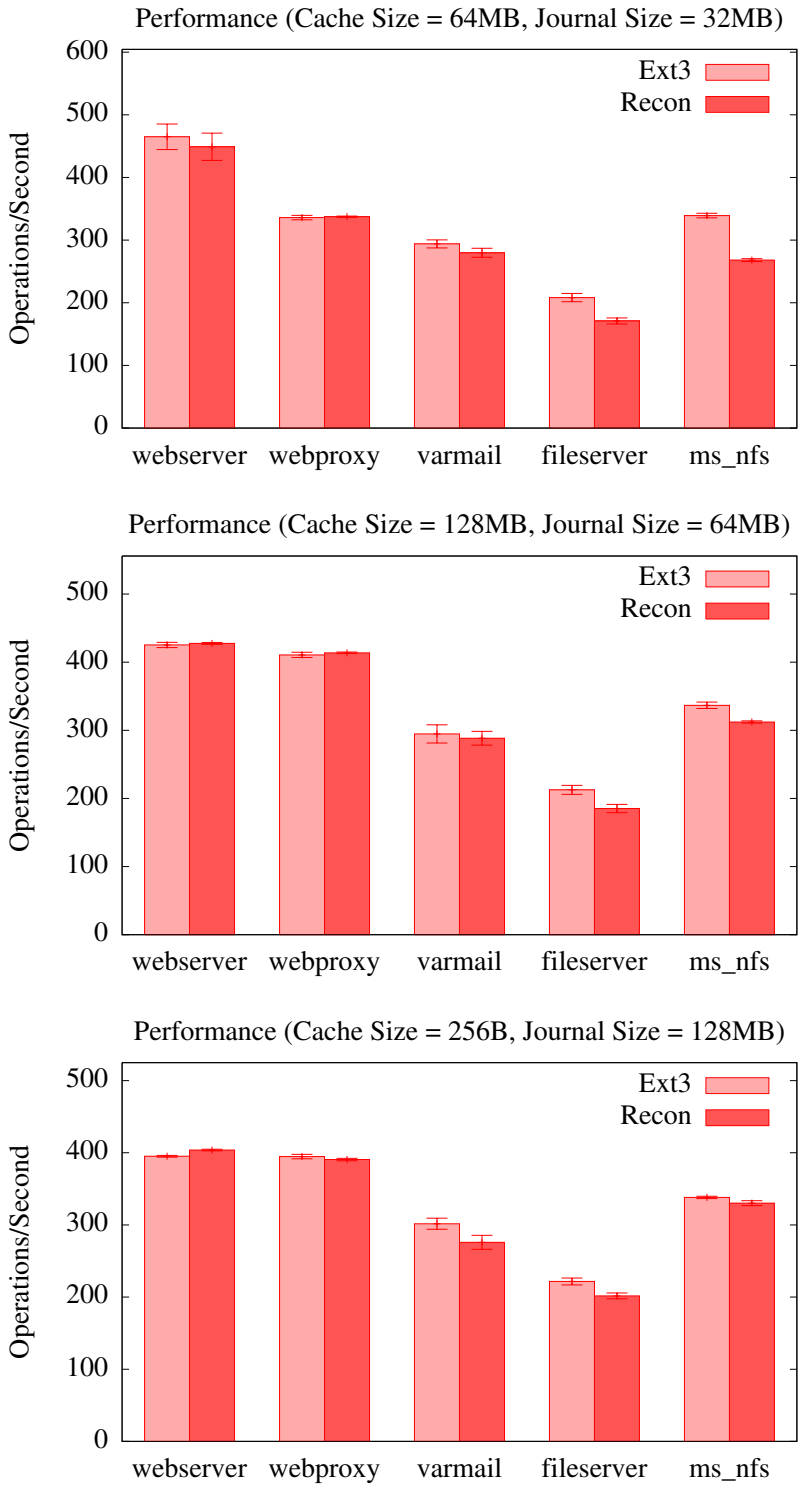


Figure 4.5: Performance on Filebench workloads for varying metadata cache sizes

Personality	Recon Cache Size	Filesystem Reads x 1000 (% Fast)	Recon Cache Misses x 1000	Total FS Read Delay	Total FS Commit Delay	Total Updated Metadata Blocks	Number of Txns
Webserver	64MB	2346.2 (0.3%)	4.2	3.4%	0.0%	1069	156
	128MB	2245.4 (0.4%)	0	2.3%	0.1%	162	11
	256MB	2073.6 (0.3%)	0	2.4%	0.1%	152	10
Webproxy	64MB	1518.8 (4.7%)	0.1	3.1%	1.7%	177987	155
	128MB	1790.1 (3.1%)	0	1.9%	1.6%	187570	91
	256MB	1668.6 (1.1%)	0	1.2%	0.9%	112408	38
Varmail	64MB	619.8 (0.7%)	6.0	4.4%	4.1%	756877	18468
	128MB	689.9 (1.1%)	0	0.2%	1.6%	791525	20909
	256MB	683.8 (1.1%)	0	0.2%	1.5%	755344	20400
Fileserver	64MB	489.0 (0.1%)	68.7	30.9%	18.3%	133877	86
	128MB	573.3 (4.6%)	31.5	5.8%	7.1%	140299	47
	256MB	622.4 (6.3%)	10.7	1.2%	1.5%	146584	30
Ms_nfs	64MB	6735.1 (0.1%)	71.5	23.7%	21.3%	359864	584
	128MB	7730.1 (0.3%)	27.0	9.4%	8.1%	401677	593
	256MB	8314.0 (0.6%)	2.4	1.3%	0.9%	420449	601

Table 4.6: Benchmark statistics for Recon

direct impact on the workload. Other workloads, like webserver, however, commit very few transactions and are seldom blocked waiting for the commit to complete.

Second, Recon can increase the number of misses in the file system buffer cache, thereby increasing the number of read requests that are sent to disk, because the memory used for Recon's caches reduces the memory available to the buffer cache. The cost of these extra reads can be mitigated somewhat by serving metadata read requests from the Recon read cache, rather than issuing them to disk (we call these "fast reads").

Table 4.6 presents these measurements for each Filebench workload at each Recon cache size. We report total file system read delay and commit delay as a percent of the workload runtime (roughly 3600 seconds). For read delay, we recorded the cumulative time added to all read requests due to Recon, which includes both the processing time, and the time waiting in the work queue for the recon thread to process the request. Since multiple threads may issue read requests concurrently, the delays experienced by different threads can overlap. We make several observations about these results.

First, we find that the number of transactions usually decreases with increasing cache size, due to the corresponding increase in the journal size. This is not the case for varmail, however, where the number of transactions is driven by the frequent sync operations in the workload, and not by a limited journal size. We also see that the number of transactions in the ms_nfs workload increases slightly with increasing cache sizes.

Second, we see that the impact on Filebench throughput is closely related to the number of misses in the Recon metadata cache. For example, fileserver and ms_nfs workloads have similar numbers of misses, and similar overall losses in throughput for each cache size. These misses occur during recon_commit processing, leading to both extra commit delays while the missing metadata is read back into the Recon read cache, and extra read delays. The impact of a miss during commit can be inflated because multiple read requests may be waiting for the recon thread to finish checking the transaction before it can process the next request on its work queue. As expected, increasing the size of the Recon cache reduces the number of misses for all workloads.

Third, we consider whether Recon's caches are increasing the number of file system buffer cache misses, but

looking at the change in the number of read requests issued by the file system as the Recon cache size increases. For webserver, webproxy, and varmail there is little evidence that Recon's caches are increasing the buffer cache miss rate. For the workloads that emulate file servers (fileserver and ms_nfs), however, we do see a steady increase in the number of read requests from the file system as the Recon cache size increases. This result suggests that the buffer cache becomes less effective, due to the memory consumed by Recon. The increase in file system reads, however, does not correlate with the overall throughput of the workload. We find that the benefit of decreasing the Recon miss rate outweighs the cost of increasing the buffer cache miss rate.

Finally, we see that serving metadata reads from the Recon cache accounts for a small percentage of the total reads issued by the filesystem. This result is expected, since there are generally many fewer metadata reads than data reads. We find that the benefit of avoiding disk access for these reads is greater than expected, however. For example, for webserver with a 256MB cache, Recon is able to satisfy only 0.3% of read requests from its metadata cache, but, Recon improves webserver throughput by 2% relative to native Ext3. We believe this is because metadata reads generally have worse locality than data reads, and thus incur greater positioning delays (seek and rotational latency).

4.5 Summary

Recon demonstrates that run-time checking can be used to enforce a full suite of file system consistency invariants with low overhead. Recon can be isolated from a potentially corrupt operating system because it uses I/O introspection to detect and interpret the file system metadata that it checks. The consistency invariants that Recon checks are derived from the global consistency properties that an offline checker would check. In Chapter 6 this thesis discusses a technique for doing this derivation more systematically, called *differential invariants*.

A significant portion of Recon's overhead is because it has to compare the old metadata state with the new state to detect changes, and because some invariants must read the old values of fields. The memory overhead is primarily due to the need to cache the old metadata separate from the file system, and the CPU cost is primarily in the code which compares the old and new states of the file system. However, for many workloads this overhead only amounts to a small drop in throughput.

Recon depends on the assumption that crash consistency mechanism of the file system is operating correctly. This is because it checks writes at a consistency point, but not otherwise. In the next chapter, we show how we can remove this assumption.

Chapter 5

Location Invariants for Durability and Atomicity

Recon takes advantage of transactional methods, such as journaling [18, 78, 81] and shadow paging [7, 38, 68], used by modern file systems for providing crash consistency. In particular, it checks that metadata updates within a transaction are mutually consistent at transaction commit time. This approach is still vulnerable to file system corruption when the transactional mechanism is used incorrectly or has bugs. For example, the Recon checker for the Ext3 journaling file system verifies writes to the journal blocks, but it assumes that 1) all metadata writes first go to the journal, and 2) these writes are then checkpointed correctly. Any bugs that violate these assumptions, e.g., a lost or failed checkpointing write, will cause undetected corruption. In Section 5.1.2, we show that these bugs manifest in many different ways, such as lost, misdirected, out-of-order and corrupting writes, making it difficult to detect them. Unfortunately, these types of bugs occur regularly [26], are hard to diagnose [30, 70], and can have serious impact [80].

In this chapter, we describe the design and implementation of a runtime checking system that enforces correct usage and implementation of the crash consistency method used by the file system. The study of bugs over the patch history of Linux file systems [52] showed that crash consistency bugs happen roughly in proportion to the size of the crash consistency mechanism code within the file system, typically 10-20%. Our system enforces the atomicity and durability properties of the file system at each block write, in addition to checking consistency at commit time, providing the strong guarantee that every block write will maintain file system consistency.

We express the atomicity and durability properties as invariants, called *location invariants* because they govern which blocks are written to given locations. We describe the location invariants that need to be enforced to preserve the integrity of committed transactions for journaling and shadow paging file systems, and the file system properties that make it feasible to check these invariants efficiently at the block layer.

We augment Recon by implementing runtime checking of location invariants for the Btrfs and Ext3 file systems. We evaluate the overhead by comparing native file system performance to the performance of Recon both with and without location invariant checks on top of consistency invariant checking.

Our evaluation shows that the runtime checkers for both the file systems detect violations of atomicity and durability effectively and efficiently. For both file systems, checking location invariants has negligible overhead on top of consistency checking. However, checking the conistency invariants for Btrfs incurs more substantial overhead than in Ext3. This is for two reasons: Btrfs updates larger quantities of metadata, so there is more to process, and the copy-on-write nature of Btrfs requires us to do more processing in order to compare the old and

new states of the file system.

5.1 Motivation

Our aim is to design a runtime checking system that can reliably detect file system and other operating system software bugs, and memory corruption errors, before they cause on-disk data corruption. Unlike an offline file system checker, a runtime checker does not detect file system corruption caused by I/O hardware failures, such as device controller failures or latent sector errors on disks. Instead, the runtime checker depends on hardware redundancy mechanisms, such as checksums and replication [64], implemented either in the storage system or in the file system [7, 68], to detect and recover from such failures when data is read from disk.

A runtime checking system can be deployed in either a development or a production setting. During development, a runtime checker can serve as a testing tool, catching subtle errors before the file system image becomes inconsistent, making it easier to determine the root cause of a bug. In production, the checker could trigger measures to preserve existing data, recover from the failure [77], or alert administrators to the problem. Our runtime checking system builds on the Recon system [26], and so this section starts by providing an overview of Recon. Then we motivate this work by discussing the types of bugs that Recon will fail to detect, leading to undetected data corruption.

5.1.1 The Recon System

The Recon system takes advantage of transactional methods, such as journaling and shadow paging, used by modern file systems for providing crash consistency. These transactional methods group writes to disk blocks from one or more operations (such as the creation of a directory and a file write) into transactions. When transactions are committed, the file system believes itself to be consistent. At this point, Recon checks that the contents of the blocks involved in the transaction are mutually consistent, thus detecting the effects of software bugs (or memory errors) that corrupt blocks within the transaction.

The consistency checks in Recon are derived from the consistency properties of the file system. These properties constrain the set of valid file system states that can be generated by an arbitrary sequence of file system operations. Typically, these properties are checked by the offline file system checker. For example, a consistency property in the Btrfs file system is that extents must not overlap. Checking this property requires a full scan of the extent tree, making it infeasible to perform at runtime. Instead, each consistency property is transformed into a local consistency invariant, which is an assertion that must hold for the transaction blocks to preserve consistency. In the Btrfs example, the consistency invariant is that when a new extent item is added to a tree, then the extent must not overlap with the previous or next extents in the extent tree. A runtime checker can enforce this consistency invariant by examining all updated extents and their adjacent extents.

The Recon system interposes at the block layer, and can be implemented in the host operating system, a hypervisor or a storage controller. The benefit of this approach is that the checker only depends on the the format and the consistency properties of the file system, rather than depending on the implementation of the file system, which may be buggy and cannot be trusted. File system formats and their consistency properties tend to be stable over time, even when the implementation changes significantly over time, or there are multiple different implementations of a particular file system.

Figure 4.1 shows the architecture of the Recon runtime checker. Recon is composed of a generic framework and file system specific components that plug in through a simple API. Since Recon interposes at the block layer,

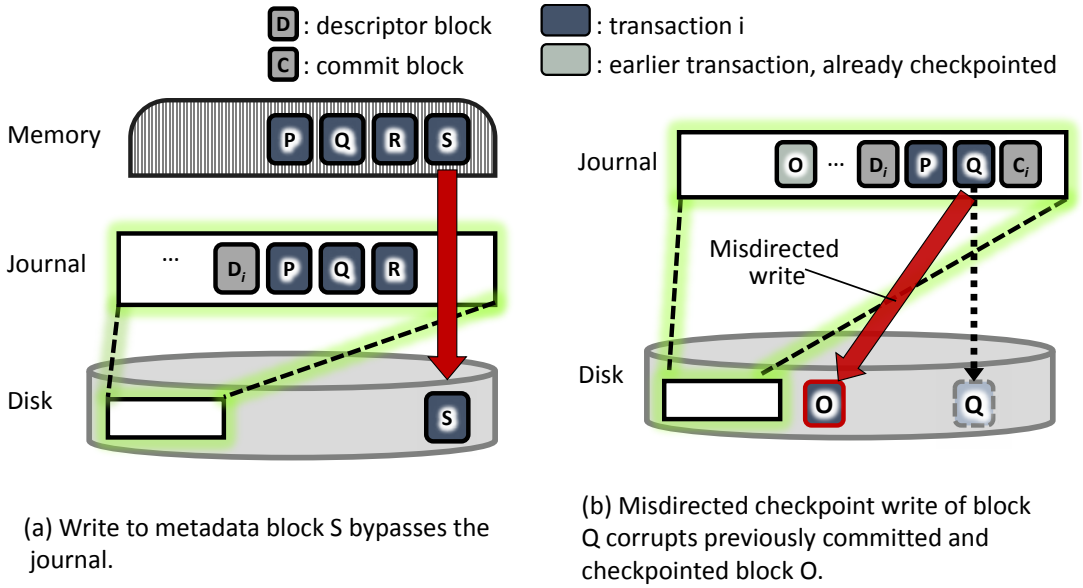


Figure 5.1: Examples of overwrite bugs in a journaling file system.

it uses an introspection approach, similar to semantically smart disks [75], to infer the types of blocks as they are accessed, and then interprets the block contents to derive the logical file-system data structures. The consistency invariants are expressed in terms of logical file-system data structures, such as the extent information in the Btrfs example.

The read cache in Figure 4.1 caches the on-disk file system state, while the write cache caches the metadata blocks updated in a transaction. During commit, Recon uses file-system specific components to compare the file-system data structures in the two caches, generating records of logical changes.

These records capture any modifications to file system objects, such as the addition of a new object, an update to a field in an existing object, or the removal of an object in a transaction. Invariant checks are triggered by change records, but the check may require additional information about objects that have not changed in the current transaction. Query primitives are used to retrieve this information from the metadata caches. The consistency invariants verify that when the logical changes are applied to the read cache, they will result in a consistent file system state. If so, the transaction commit is allowed, and then the contents of the write cache are merged into the read cache, updating the checker's view of on-disk state, and the write cache is cleared.

5.1.2 Problematic Bugs

Recon ensures that the blocks in a transaction are consistent, but it depends on the transaction mechanism being both implemented and used correctly. Below, we describe four classes of bugs that break these assumptions, and provide some examples of recent bugs in the Ext3, Ext4 and the Btrfs file system code deployed in "stable" Linux kernel releases.

Overwrite bugs: A write occurs to a location when it shouldn't have happened at all, either due to improper writing or flushing of buffers, or some other failure that causes a misdirected write. For example, Ext4 stores file system quota information as data in special quota files. The contents of these files are metadata, similar to directories, but they were overwritten in place without first writing to the journal, when the file system was used with certain mount options [44]. Recon's consistency invariants would not detect this problem because

the journal would appear to be consistent. Similarly, a high profile bug was recently introduced in the Ext4 file system, in which the inode bitmap was modified without updating the journal, which could lead to occasional corruption [70]. Interestingly, after the corruption issue was reported, the developers at first mistakenly thought that the root cause was an incorrect update to the journal superblock [80]. This suggests that understanding, using and implementing the transactional mechanism is challenging and bug prone. In this case, if the file system is allowed to continue running, the transaction that was missing the inode bitmap update in the journal would commit, and the checkpoint of that transaction would bring everything back to a consistent state, with no one the wiser. Consistency problems only occur when an ill-timed crash forces recovery from the incomplete journal entries. When Recon is used in production, things actually become worse. Recon would detect that the journal contents are inconsistent, because the inode bitmap updates are missing (e.g., unallocated inodes would appear to change), and then discard the transaction and stop the file system. The inode bitmap, overwritten in place, would cause the file system to become inconsistent. Figure 5.1 illustrates two of these forms of overwrite bugs. In Figure 5.1(a), the metadata block S is being written directly to its final destination without first updating the journal, similar to the Ext4 inode bitmap bug. In Figure 5.1(b) metadata block Q has been successfully committed to the journal, but the subsequent write of that block to its final location is misdirected, corrupting a previously committed (and checkpointed) metadata block O and leaving an older, inconsistent version of Q on the disk.

Lost write bugs: A write that should happen doesn't occur. For example, in a journaling file system, a lost checkpointing or recovery write will cause file system inconsistency even though the journal is consistent [32].

Write ordering bugs: The file system needs to enforce ordering of writes to disk at certain times. While the block layer may observe writes in the correct order, unless the correct disk barrier commands are sent, the disk or its controller may reorder writes, causing inconsistency of the on-disk state on a power loss. For example, Linux JBD2 journaling code maintains a pointer to the journal tail in a journal superblock. When the tail was updated, the journal superblock was not being flushed to disk before new transactions could reuse the newly freed journal space. On a power loss, the recovery code could replay old transactions containing blocks potentially overwritten in the journal by new transactions [45], including blocks from uncommitted transactions. Similarly, the Btrfs file system in multi-device setups (e.g., mirroring) would send barriers in the wrong order and not wait for all the barriers before writing the commit block [55]. These write ordering bugs would not be detected by Recon but they can cause serious file system inconsistencies.

Corrupting Write bugs: A write occurs to the correct location but its contents are corrupt. For example, the Ext3 journaling code modifies (escapes) its data blocks when they start with a magic number that identifies journal metadata blocks, to distinguish between the two types of blocks. When Ext3 was used in data journaling mode, the recovery code had a bug that would unescape[1] the wrong buffers, causing corruption of both the block that remains escaped, and the block that is wrongly unescaped [30]. This bug would not be caught by Recon's consistency invariants because the journal itself is not corrupt. However, blocks from committed transactions would be corrupt on disk following recovery.

5.2 Location Invariants

File systems that use transactional mechanisms for crash consistency provide atomicity and durability properties. Atomicity properties ensure that the file system will be able to roll back to a consistent state on a crash. Durability

[1] That is, restore the magic number at the beginning of the buffer

properties ensure that if a new version of a block is committed, it does not get rolled back or overwritten, except atomically as part of a subsequent transaction.

The problematic bugs described in Section 5.1.2 can cause corruption because they lead to violations of these properties. For example, a metadata overwrite that is not first committed to the journal violates atomicity, since we cannot roll back to the previous correct version of the block. Durability can be violated by either an omitted checkpoint write, or a write that corrupts a committed transaction in the journal, since updates that were successfully committed to the journal never reach the file system. Finally, in both journaling or shadow paging systems, a misdirected write that overwrites an allocated metadata block (e.g., a data block write that overwrites a metadata block) violates both atomicity and durability.

In this section, we first describe what is needed to detect violations of these properties, and then present the location invariants for journaling and shadow paging transactional mechanisms.

5.2.1 Enforcing Atomicity and Durability

Checking the consistency of metadata as it is committed to a journal or copy-on-write tree is not sufficient to guarantee that the metadata will be stored atomically or durably. Originally, Recon depended on the correctness of the file system's transactional mechanism to properly enforce the atomicity and durability of the metadata updates that it is checking. This assumption can be violated by several classes of bugs, as explained in Section 5.1.2.

To detect these bugs, a runtime checker needs to enforce atomicity and durability invariants, in addition to consistency invariants. Consistency invariants apply to the contents of updated blocks; they need to be checked at transaction commit points because the file system does not guarantee that the updates are consistent until the commit. In contrast, the atomicity and durability invariants need to be checked on each block write, because they govern whether the write is permitted to the given location. Hence, we call them *location* invariants collectively. Rather than being derived from the offline checking tool, the location invariants are derived from the semantics of the transactional mechanism itself. In particular, they concern *overwrites* to the blocks, and the *ordering* of block write operations.

It is possible to enforce both atomicity and durability invariants on each write because they only depend on the correctness of committed metadata, which has already been checked using consistency invariants. Transactional techniques like journaling or shadow paging must first write metadata to unallocated blocks – for journaling, these are free blocks in the journal area, which must later be checkpointed back to the file system, while for shadow paging these may be any free blocks, which become part of the file system atomically at the commit point. To check that these properties are maintained, location invariants depend on information about block allocation and block type (data vs. metadata). The block allocation information must be based on committed metadata, since uncommitted changes to the allocation state may be rolled back following a crash. In particular, we must not permit a write to a block that has been freed in an uncommitted transaction, since we would not be able to recover the previous version of the block if the deallocation operation were rolled back.

As can be seen, correct checking of consistency and location invariants is interdependent. We begin from the assumption that the file system state on disk is consistent. Initially, this is the result of correct file system initialization, as is done by mkfs. Thereafter, each block write prior to a transaction commit is checked by the location invariants using the old, consistent, committed allocation and block type information. These checks ensure that the committed state is not corrupted. At the transaction commit point, the contents of the transaction are checked by the consistency invariants to ensure that the new file system state will be consistent. The location invariants then govern the write of the commit block itself, and the subsequent checkpoint writes to the file system, as well as the writes of blocks in the next transaction. By enforcing both consistency and location invariants, the

runtime checker can provide the strong guarantee that the file system meets its consistency specification on every block write.[2] As we will see in the next subsection, there are significant differences between the specific location invariants that apply to journaling and shadow paging mechanisms. However, both require the ability to infer block allocation information and the ability to distinguish between metadata and data blocks at the block layer.

5.2.2 Journaling Invariants - Redo Logging

Journaling file systems use write-ahead logging to support failure atomicity. First, they write a consistent set of blocks and their final location information to a designated journal area. When all these blocks are durable in the journal, an atomic journal write signals a commit. After commit, the contents of the journal are flushed to their final locations. This flush to the final file system locations is called checkpointing in the Linux Ext3/jbd terminology.

The journal area must be known to the runtime checker so that, on each write, it can distinguish between journal and non-journal writes. This distinction is necessary so that the correctness of both the journal writes and the checkpointing writes can be verified. Checkpointing of committed transactions occurs concurrently with new journal writes, but checkpointing writes must be directed to the non-journal area. Note that although we expect the journal to be a circular buffer, with writes occurring sequentially, at the block layer there is no guarantee of any particular ordering within a transaction.

The following four location invariants ensure that the journaling and checkpointing operations of the file system are correct:

1. Log invariant: A write to the journal area must be to a free block in the journal. A free journal block becomes allocated when it is written and free again when it has been checkpointed (see Checkpoint invariant below). This invariant checks that the allocated journal blocks are not overwritten.

2. Commit invariant: A write of a commit block, which marks a transaction as committed, is allowed to the journal area only after (1) all the blocks in the transaction are allocated in the journal, and (2) a barrier is issued to flush these transaction blocks to the disk. The transaction is considered to be committed (and hence, to be durable) only after the commit block is flushed to disk. When journal checksums are included in the commit block, as in IRON file systems [64], the write of the commit block can be concurrent with the writes of the transaction blocks, but a barrier is still needed to ensure that all these blocks are on disk before the transaction is deemed to be committed.

3. Flush invariant: A write to an allocated, non-journal location is permitted only when: (1) the committed part of the journal contains a block that is destined for the same final location, and (2) the contents of the written block match the contents of this block in the journal.

   In other words, overwrites of allocated non-journal blocks are disallowed if the new content was not first committed to the journal. If the block exists only in the uncommitted portion of the journal, or the block does not appear in the journal at all, both atomicity and durability violations can occur. Atomicity is violated by writing new content into the file system ahead of the commit of the transaction that should contain it. Durability is violated by the loss of previously committed content that has been overwritten.

[2] While the checker implementation may have bugs that generate false alarms, it is unlikely that the checker will fail to detect file system corruption, unless its bugs are correlated with file system bugs [26].

4. Checkpoint invariant: A write of a checkpoint record (e.g., in the journal superblock), which indicates that a set of blocks in the journal area are now free, is permitted only after all the journal blocks for the associated transaction have been either (1) flushed (see Flush invariant), or (2) superseded by a newer version of the corresponding block in a later committed transaction. If a newer version of a block exists in a later committed transaction in the journal, then this version does not need to be flushed before being freed. The affected journal blocks can only be considered free after the checkpoint record has been flushed to disk.

Metadata-only Journaling Since writing to the journal potentially doubles the total write traffic to disk, many file systems allow journaling only metadata blocks to reduce write traffic. The main complication with metadata-only journaling is that data writes are non-atomic, and while these writes must be allowed at any time, they must not overwrite metadata blocks. To accommodate non-journaled data writes, we refine the journaling flush invariant with an exception:

1. Data-flush exception: Any non-journal write that violates the flush invariant must be to a non-metadata (data or free) block location. The type of a block (metadata or not) is determined by the committed file-system state. The consequence of this exception is that data writes can overwrite data blocks unimpeded. Unfortunately, there is no way to tell if data writes are misdirected among each other.

The challenge with allowing this exception is that it must be possible to distinguish metadata blocks from non-metadata blocks on each write, but a file system may not provide this information easily. For example, the Ext3 file system uses allocation bitmaps that allow distinguishing between allocated blocks (which may be data or metadata) and free blocks. However, the file system does not provide an easy way to distinguish between dynamically allocated metadata (e.g., for directories and indirect blocks) and data blocks, other than by traversing the entire file system. We discuss this issue further in Section 5.3.3.

5.2.3 Shadow Paging Invariants

Compared to journaling, it is simpler to enforce location invariants for shadow paging systems because blocks are updated once per transaction and all these updates occur before commit. In a file system that uses shadow paging for all blocks, there are two atomicity invariants:

1. Flush invariant: All writes, other than to special non-shadow paged blocks, such as the super block, must be to unallocated blocks. This invariant follows from the basic copy-on-write properties of shadow paging systems. To enforce this invariant, the file system must provide an efficient method for determining the allocation status of a block. For example, the Btrfs file system maintains an extent allocation tree.

2. Commit invariant: The write of the commit block (usually a tree root) is flushed to disk only after both (1) all blocks referenced by the new tree have been updated, and (2) a barrier is issued to flush these blocks to disk. That is, there must be no dangling pointers to potentially uninitialized blocks, before the commit block is flushed.

Durability (e.g., a lost or corrupting update) is checked in modern shadow paging file systems using methods such as block checksums (ZFS) or generation numbers (Btrfs). This information is embedded in metadata blocks, and hence our Btrfs runtime checker uses consistency invariants to check the consistency of block headers and generation numbers for ensuring durability.

Metadata-only Shadow Paging Shadow paging can lead to fragmentation because the updated blocks are placed in new, possibly distant, physical locations. Fragmentation can be reduced with metadata-only shadow paging, with data writes being performed in place. To accommodate non-atomic data writes, we refine the flush invariant with an exception:

1. Data-flush exception: Any write that violates the flush invariant must be to a non-metadata (data or free) block location.

This exception requires being able to distinguish metadata and non-metadata blocks. Btrfs tracks whether an extent has metadata or data in its allocation tree, making it easy to enforce this invariant. Also, the default behaviour of Btrfs is to separate metadata and data regions, making this identification even easier and more efficient.

5.3 Implementation

As explained in Section 5.2.1, location and consistency invariants are interdependent, and they need to be checked together. Hence, we have implemented location invariant checking for the Linux Ext3 (journal invariants) and Btrfs (shadow paging invariants) file systems by augmenting the Recon consistency checking system. Recon uses the block-layer Linux device mapper framework to interpose on block I/O, allowing location invariants to be checked on all writes. The block-layer approach ensures the independence of the checker and the file-system implementations. Next, we describe the requirements for implementing a runtime checker, and then discuss how these requirements are met in our implementation.

5.3.1 Runtime Checker Requirements

File system design impacts the capabilities and performance of a runtime checking system. In this section, we present the four types of information needed by a checker. The challenge is to obtain this information correctly and efficiently at the block layer. The more file system state that must be examined to do so, the higher the overhead of the checker.

Consistency Points: Runtime checking at the block layer requires being able to get a consistent picture of the file system state from outside the file system. Consistency points provide both a point in time to check consistency invariants and a consistent view of the file system when checking location invariants on each write.

Allocation Information: A checker needs to distinguish between allocated and unallocated blocks, particularly on the write path, to protect against accidental overwrites. Overwriting an unallocated block is harmless, but location invariants constrain when allocated metadata blocks can be overwritten.

Separate Metadata: The checker also needs to distinguish between metadata and data blocks on both the read and the write paths. Metadata blocks are cached to improve checker performance, since recently accessed metadata is likely to be relevant to invariant checking, while data blocks are ignored because they are not interpreted. Additionally, the location invariants may permit or forbid a write depending on whether the destination is a data or metadata block.

Block Identity: Finally, interpreting a metadata block requires knowing the *identity* of the block. The block identity determines the logical contents of the block in the file system. For example, suppose that the checker knows that some block is an inode block, and it identifies the block as the fourth inode block in the file system. If it knows

that inode blocks contain 32 inodes, then it can determine that this block contains inodes with numbers 97-128. A runtime checker can then correlate these inodes with directory entries that reference them, with inode bitmaps that allocate them, and with the indirect blocks to which they point. Without knowing their specific identities, it would not be possible to make the associations between the data structures that are needed for enforcement of invariants.

5.3.2 Block-Layer Metadata Interpretation

In this section, we discuss two complementary approaches for determining block identity. The following sections describe how we apply them to interpret metadata in the Ext3 and Btrfs checkers.

Forward Pointers: File systems are tree structures or directed acyclic graph structures, with parent blocks containing some form of a pointer to child blocks. Thus, the easiest way to identify a block is if we are already traversing the parent block. For example, if the checker (or the file system) is looking up some specific metadata, starting from the root of the tree, it can traverse intermediate blocks to locate the desired block.

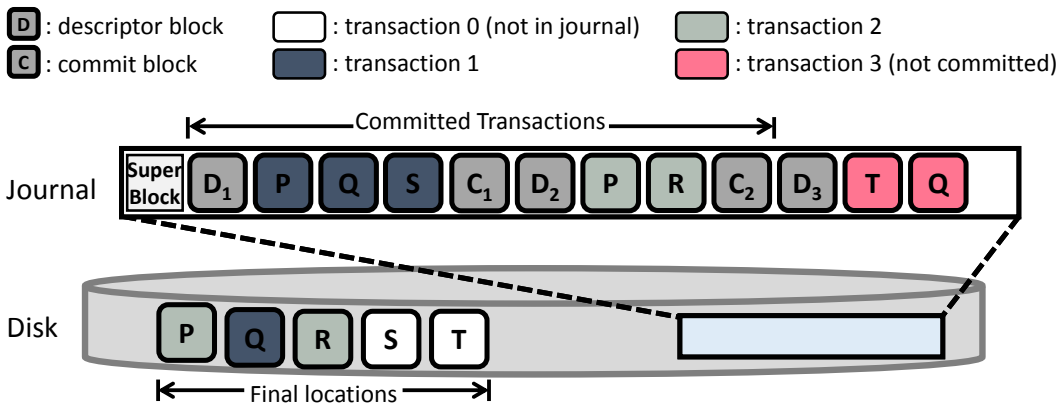
Back References: A back reference for a block is metadata that maps the block's physical location to blocks that reference the block [54], providing an efficient method for locating parent blocks. Back references are used for various tasks such as defragmentation and bad block replacement, in which the parent block containing the reference must be efficiently located and updated. The parent block has information to help type and identify the child block, and hence back references greatly simplify metadata interpretation. However, looking up a back reference may incur additional I/O operations.

5.3.3 Ext3 Implementation

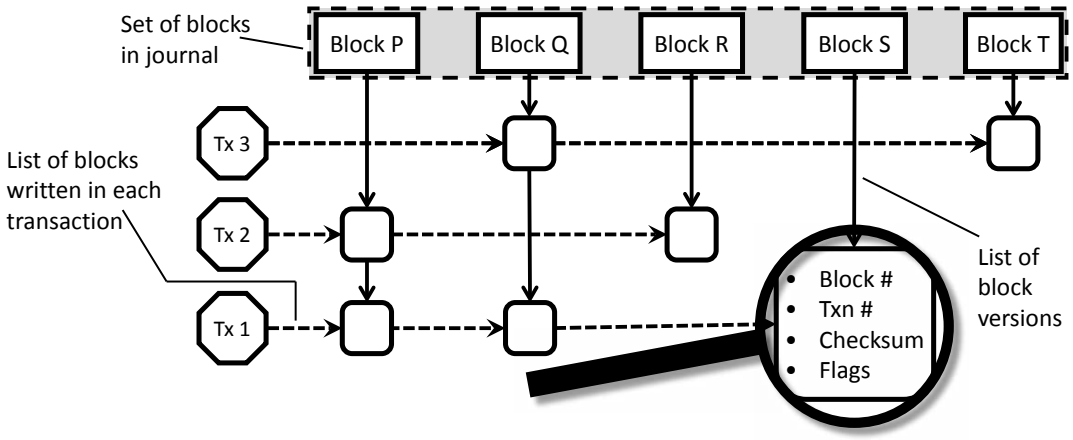
Ext3 uses static block allocation bitmaps, making it easy for the checker to determine the allocation status of blocks. However, Ext3 does not provide any efficient method for distinguishing metadata blocks from other blocks, either on block writes or on block reads that violate pointer-before-block traversal. Instead, we retrofit the Ext3 file system with a metadata allocation bitmap, which records whether a given block is metadata. The new metadata bitmap is stored alongside the block allocation bitmap. Using the metadata bitmap, the checker can ensure that data blocks are never cached on either a read or a write, and the data flush exception, described in Section 5.2.2, can be implemented easily. Without modifying the file system, an alternative is to track all data and metadata pointers that have been seen, in order to ensure that the type of a block is known before that block is written to. This approach would incur a high memory overhead.

Interpreting Metadata

The Ext3 file system does not provide back references. Instead, we use the file system's forward pointer traversal to create in-memory back references dynamically. The file system needs to read the parent of a block at least once before it accesses the child block, which we call pointer-before-block traversal. When the parent block is read the first time, we create a back reference for each of the child blocks to which it points. For example, when an inode block is read by Ext3, we copy the block into the read cache, parse the inodes in the block, and then create back references for all child metadata blocks (e.g., indirect blocks) directly pointed to by the inodes. The back reference contains the block type, and for an indirect block, it contains the inode number and an offset that locates the indirect block. When the indirect block is read, its back reference will exist, and hence the block can be typed and identified. These back references are bootstrapped using the superblock, which exists at a known location.



(a) Blocks in the journal at a particular point in time, belonging to multiple transactions.



(b) Data structures used to check location invariants on every file system write to disk.

Figure 5.2: Checking location invariants in Ext3.

The main drawback of in-memory back references is that they cannot be evicted because the file system may cache information from the parent block indefinitely, allowing it to access the child block directly at any time in the future. However, the in-memory references could be persisted by leveraging the backpointer-based consistency techniques developed in NoFS [13] and ffsck [53].

Location Invariants

The Ext3 location invariants require tracking the state of the journal. We refer to physical storage blocks allocated to the journal as *slots*, each of which may contain either a version of a metadata block that has been written into the journal, or journal control structures such as descriptor blocks and the commit blocks that end a transaction. A slot in the journal can be in one of four states: logged, committed, flushed, and free. These four states correspond to the four journaling invariants described in Section 5.2.2. Note that a slot stays allocated (as explained in the Log invariant) during the logged, committed, and flushed states.

The checker maintains three data structures: a list of transactions currently present in the journal, an array containing information about the status of each journal slot, including block checksums, and a hash table mapping

from metadata block numbers to versions of that block in the journal.

Figure 5.2(a) shows an example set of blocks in the journal, including the descriptor and commit blocks and the journaled metadata blocks themselves. Figure 5.2(b) shows the checker's data structures for this set of transactions. The octagons on the left represent the list of transactions. For each transaction, we build a list of the slots containing blocks written in that transaction, shown by the dashed arrows in the figure. The hash table is shown as the shaded rectangle at the top, with the solid arrows representing a linked list of versions of each block in the journal.

Based on writes to different types of journal blocks (i.e., the descriptor blocks, metadata blocks, commit blocks, and the journal superblock) and non-journal blocks, the checker updates its data structures and the block states, and enforces the journaling invariants described in Section 5.2.2. For example, when we see a write to a metadata block we can use the hash table to locate committed versions of that block in the journal and verify that it matches one of them. If it does not, this indicates a violation of the Flush invariant.

During a commit, if a new metadata pointer is found without a corresponding new metadata block in the journal, we detect a violation of the commit invariant (that all blocks should have been written before commit).

One complication with metadata-only journaling is that Ext3 uses *revoke* records to indicate that a metadata block has been freed, and could be reused as a data block that is updated non-atomically. As a result, any versions of this block in previous transactions should no longer be checkpointed or else the data block could be overwritten. The checker handles such revoked blocks by marking their status as checkpointed, so that the Checkpoint Invariant does not fail if the containing transaction is freed without seeing a write to that block.

5.3.4 Btrfs Implementation

Btrfs provides various features such as extent-based allocation (which allows a single allocation record to cover multiple blocks), back references (which help tasks like online defragmentation) and writable snapshots (which are isolated from the original version using copy-on-write semantics). Btrfs uses shadow paging for ensuring crash consistency, similar to the WAFL file system [38].

Btrfs uses multiple B-trees to store its metadata. A root B-tree contains pointers to the roots of other B-trees, including the main file system tree, snapshot trees, and an extent tree that records allocation information. Each B-tree consists of internal nodes and leaves. Internal nodes contain an array of key/block-pointer pairs, with the key representing the smallest key stored in the pointed-to node or leaf, and the block pointer helping locate the child node or leaf on disk. All Btrfs metadata blocks begin with a header that has a block checksum, a generation number, and the id of the tree containing the block.

We found that Btrfs can issue writes from concurrent transactions. For example, blocks from the next transaction can be written to disk before the current transaction commit, but as expected, the next transaction blocks are unreachable from the current transaction. As a result, the Btrfs checker assumes that unreachable blocks belong to a future transaction and delays processing them.

Interpreting Metadata

Btrfs uses shadow paging, so that when a leaf node is updated, all its ancestor nodes are also updated. Because of this property, the checker can use forward pointer traversal on commit, starting from the superblock.

Btrfs uses an extent B-tree to store allocation information, which the checker also uses to determine the allocation status of blocks. Similarly, separating data and metadata blocks on both the read and write paths is relatively easy because Btrfs allocates separate large contiguous regions for data and metadata. However, if Btrfs

is operating in a "mixed" region mode (not a common configuration), data extents can be distinguished from metadata extents by traversing the extent allocation tree and examining the per-extent flags.

Btrfs uses typed and self-identifying metadata blocks. Each metadata block has a header that stores the type (node or leaf) and level of the block in the tree, and the first key in the block is its identity, helping locate the block in the tree. Btrfs also supports back references to multiple snapshots, storing them with the allocation information in the extent tree.

Both back references and self-identifying metadata blocks can be used independently to type and identify blocks. We initially decided to implement a Btrfs runtime checker because we thought that both of these properties would be useful for the runtime checker. However, neither are necessary due to the forward pointer traversal enabled by shadow paging.

Location Invariants

The checker ensures atomicity and durability by checking that allocated blocks are never overwritten, which requires looking up the extent allocation tree on each write. For metadata-only shadow paging, a metadata flag in the extent record is checked to implement the data-flush exception. While checking a transaction for consistency, an invariant is tripped if a pointer to an unwritten block is encountered within the updated tree.

5.4 Evaluation

We evaluate our runtime checker in terms of its ability to detect violations of the location invariants, listed in Section 5.2, and the performance impact of checking location invariants in addition to consistency invariants for the Ext3 and Btrfs file systems. We have implemented the runtime checkers within the Linux kernel using the Recon framework, based on the approach described in the previous sections. Recon was initially developed on the device-mapper interface for Linux 2.6.27, which did not support passing disk barrier and flush requests. Recon's support for barriers in later kernels is still experimental. Our Btrfs implementation is based on the Linux 2.6.35 kernel. Recon for Ext3 is implemented and tested on Linux 3.8.11. Rather than using our modified version of Ext3 with a metadata bitmap, we use the Ext4 subsystem's emulation of Ext3, using metadata-only journaling. In our test environment, all pointers to metadata are seen while the filesystem is being populated, and so the invariants can be enforced without referring to the metadata bitmap.

5.4.1 Correctness

We evaluate the ability of our runtime checker to detect the types of bugs described in Section 5.1.2. Specifically, we inject errors into write operations issued to the block layer that result in lost, misdirected, or corrupted writes. We refer to these injected errors as corruptions. If writes are correctly ordered, and no writes are lost, misdirected, or corrupt, then the transaction mechanism is working correctly. By deliberately altering writes to violate these properties, we can evaluate whether the location invariants can successfully protect the file system.

Our corruptor sits between the file system and the checking system, and has the opportunity to act before each write is visible to the checker. The actions the corruptor can take are: 1) discard a write (lost), 2) alter the destination of the write (misdirect), or 3) alter a range of bytes within the block being written (content). Because the location invariants distinguish between several different types of blocks, we perform corruption in a type-specific manner to increase our coverage of possible scenarios and to help explain any uncaught corruption. The

Target Block Type	**Corruption Type**		
Journal Blocks	Lost	Misdirect	Content
Descriptor	10	10	8
Commit	10	10	4
Revoke	10	10	4
Metadata	10	10	2
Non-Journal Blocks			
Metadata	3	10	10
Data	N/A	10	N/A

Table 5.1: Corruptions detected by location invariants

type of a block is determined by its destination, and in the case of certain journal blocks, by the journal header stored at the beginning of the block.

Corrupting Ext3

The corruptor can target one of four types of journal blocks (journal metadata such as a descriptor block, revoke, and commit blocks, and journaled file system metadata block), or the two types of non-journal blocks (file system metadata and data). To misdirect writes, it must distinguish between free and allocated journal space, and data and metadata locations outside the journal. As Ext3 doesn't support easy metadata/data distinction on the write path, we can only target metadata blocks that we have seen pointers to, although this eventually converges on all the metadata in the file system. When the corruptor targets a non-journal metadata block write, it is emulating a bug that corrupts the checkpoint write of that metadata block. When the corruptor targets a data block write, it always misdirects the write to a non-journal metadata block, as misdirecting to another data block is undetectable when using metadata-only journaling. Likewise, lost write and content corruption types are not applied to data block writes.

Some corruptions may not violate location invariants immediately. Instead, they may lead to a future operation causing metadata corruption. For example, a lost write to the journal cannot be detected when it is dropped, and the resulting transaction may still be consistent, but the problem should be detected when the checkpoint write targets a metadata location that has not been committed to the journal. There are four distinct points in time when a corruption may be detected: during the corrupted write, at the next commit point, during the checkpoint of a corrupted transaction, and during transaction free. Any corruption that occurred in the past must be caught before a write harms the atomicity, durability, or consistency of metadata on disk.

There are a total of 16 combinations of target block types and corruption types, as shown in Table 5.1. We perform 10 corruptions per combination. Out of 160 corruptions, 131 were detected by the location invariants and 7 were detected by the consistency invariants (all 7 were content corruptions of metadata blocks in the journal). We analyzed the remaining 22 corruptions that did not trigger any invariant violations. There are two situations in which we miss corruption events, but the "corruptions" do not affect file system integrity. In the case of random content corruptions to journal metadata, much of the space in the block is unused and corruptions to the unused area have no effect on the block semantics. Together, these cases account for 14 of the missed corruptions. Similarly, when unused space in a journaled metadata block is corrupted, which occurred in one case, no invariants are violated. We verified that the corrupted space was unused by logging the range of bytes corrupted and examining the target blocks. The final 7 missed corruptions were all lost checkpoint writes. In each case, we verified that these writes were safe to omit because there was already a newer version of the block committed to the journal. In all the 22 cases where we didn't catch the corruption, the e2fsck offline checker also reported that the file system

was consistent.

Corrupting Btrfs

Testing the Btrfs location invariants is less involved, since the invariants are simpler, as described in Section 5.2.3. A buggy write in Btrfs can be redirected to overwrite an existing data or metadata block, lost, or redirected to the wrong free block. We simulated metadata blocks being misdirected by the file system by changing the block's header to match the new, incorrect location, and updating the block checksum accordingly, before feeding the block to Recon. Our checker always detected misdirections that cause overwrites of allocated data or metadata. Lost writes or writes that are misdirected to an incorrect free block are always detected by Recon during transaction processing, when a new pointer is found to a block that is missing from its write cache [26]. Lost superblock writes can potentially go undetected unless the lack of commit causes future operations to be treated as invalid. One way to mitigate this would be to check correspondence between sync() operations and a commit that Recon observes. Since system calls themselves are not visible at the block layer, this type of check is outside the current scope of Recon.

5.4.2 Performance

Setup

To measure Recon's overhead, we select three workload profiles with different behaviours from the Filebench workload generator [24]. We modify the workloads from their original versions in order to provide more realistic working set sizes for modern storage systems. The varmail profile performs many small, synchronous writes on a set of 250,000 files. The webserver profile reads many small files concurrently (100 threads) in a large directory hierarchy (approximately 250,000 files, on average 4 levels deep), while appending to a log. The ms_nfs profile simulates a single-threaded network file server, operating on a file system (100,000 files) with a file size distribution from a study of Windows desktops [58]. Our ms_nfs workload does not throttle the request rate, unlike the networkfs profile that it was originally based on, in order to measure maximum system performance. Finally, as a fourth distinct type of workload, we also measure the time it takes to create the file sets used in the benchmarks. This provides a metadata-write intensive workload which is asynchronous (in contrast to varmail), stressing write throughput rather than latency. No data set fits entirely in the buffer cache.

All benchmarks were run on a dual-core 3.0 Ghz Xeon server with 3GB of RAM. We allocated 256 MB of memory to the Recon caches, which is sufficient to cache the file system metadata for all benchmarks. The performance results account for Recon's memory usage because Linux implements a shared page cache, and so with Recon, this memory is not available to the file system cache. File systems were mounted with the 'noatime' option enabled, to clearly distinguish read intensive workloads from metadata write workloads. Specifically, the webserver benchmark generates high volumes of metadata write traffic without this option.

Hard Disk Drive

Our first experiments were run on a 250GB 7200rpm SATA drive. Figures 5.3 and 5.4 show the benchmark throughput and the time to initialize the benchmark's file system tree (setup time) averaged over 5 runs, for the Ext3 and the Btrfs file systems. Each graph shows the performance of the native file system, the file system with consistency checking enabled (Recon), and the file system with consistency and location checking enabled (Recon+AD). The throughput graphs measure the benchmark performance in operations per second, where higher

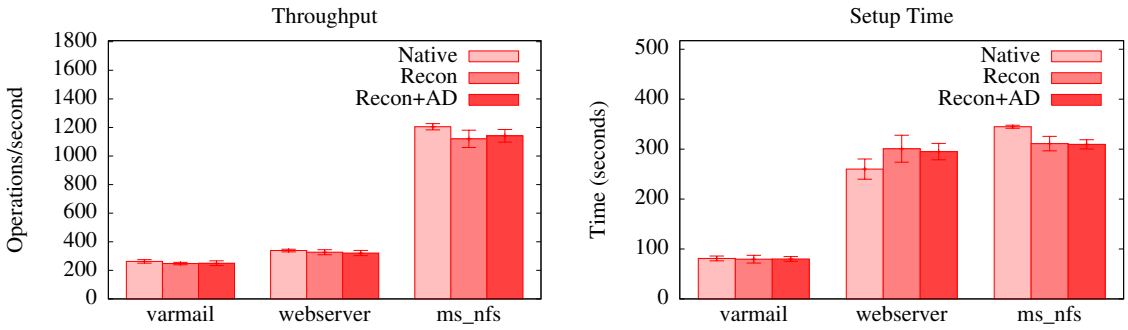


Figure 5.3: Performance on FileBench workloads for Ext3 on HDD

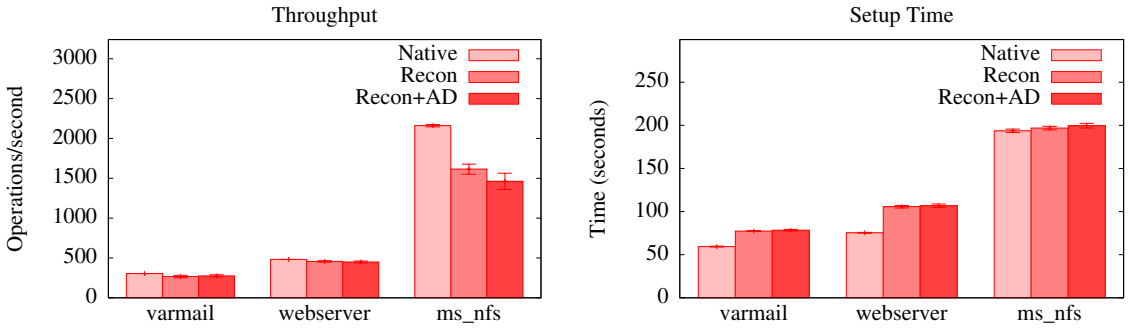


Figure 5.4: Performance on FileBench workloads for Btrfs on HDD

is better. The setup time figures are in seconds, where lower is better. These figures show that the overhead of checking location invariants is minimal compared to the existing overhead of checking consistency invariants in Recon. Even though the location invariants require a check on every write, this check is usually quick because it takes advantage of the cached metadata.

For the ms_nfs workload, Recon (both with and without location invariants) causes noticeable overhead because Recon's additional memory usage increases the pressure on the page cache. Recon's memory overhead comes from its read cache, which is limited to 256MB in between transaction commits, the write cache, which is the size of the metadata in the current transaction, and also from Recon's data structures during transaction processing. Transaction processing overheads are particularly high in Recon for Btrfs, because it constructs sets of all the items in the new and old versions of the metadata tree, in order to find the new, changed, and deleted items by set intersection. We observed that the Btrfs-recon memory requirements for these sets during transaction processing could exceed 100MB. Recon for Ext3 does not explicitly construct these sets, but instead generates the new/changed/deleted items as it directly compares the new version of a block to the old one. This direct comparison is possible because Ext3's metadata is updated in place. A possible optimization to Recon for Btrfs is to consider leaves in key-sorted order, which would limit the number of un-processed items in the "old" and "new" sets to those contained in a single leaf. Another difference between Btrfs and Ext3 is that Btrfs's transaction commits tend to be less frequent but larger. Tuning the transaction size and frequency might reduce some of Recon's impact.

Recon has a greater impact on the setup times for the varmail and webserver workloads than for ms_nfs because they involve larger numbers of small files than the ms_nfs workload; the higher ratio of metadata to data

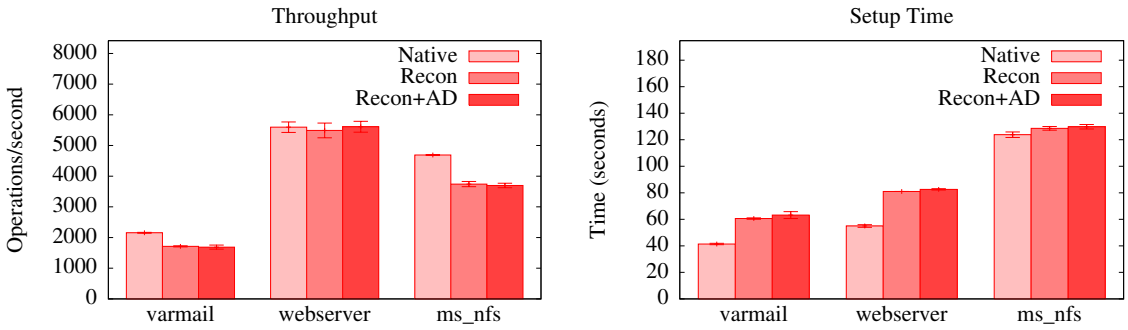


Figure 5.5: Performance on FileBench workloads for Btrfs on SSD

writes is more taxing for Recon. Since the metadata caches are of sufficient size, the overhead in this case is due to CPU time spent in transaction checking.

Solid State Drive

The relatively high I/O latencies of spinning media affect the impact of Recon's overhead. In order to investigate this impact further, we ran the Btrfs experiments on a 256GB Intel 510 SSD. Figure 5.5 compares native SSD performance to Recon interposing between the SSD and file system.

On the SSD, Recon pays an approximately 20% overhead for the varmail workload, which calls fsync frequently. This overhead is caused by Recon's CPU usage, because the fsync operations must wait for Recon to complete processing before returning. When running varmail, the CPU time used by Recon+AD was approximately 160 seconds spent on consistency checking and 38 seconds spent on location invariants, over a 20 minute benchmark. This represents an approximately 80/20 split between consistency and location invariant checking for varmail, but it is only the time spent on the consistency invariants that affects fsync() latency, as long as the CPU is not saturated. While the other workloads have similar CPU costs (in total, 45s for webserver and 129s for ms_nfs), they are not as sensitive to commit latency. Experiments with the Linux perf tool suggest that there are several effective ways to reduce Recon's CPU overhead, including more efficient implementations of hash functions and avoiding many calls to kmalloc/kfree for tiny allocations.

One aspect of fast SSD I/O is that it decreases Recon's overhead on the ms_nfs workload. While there is still a higher miss rate because of Recon's memory pressure, the page cache misses are much less costly on an SSD than on a spinning disk.

When we first ran the Btrfs benchmarks on an SSD, the webserver benchmark took a significant performance hit when running with Recon. Profiling the behaviour revealed that Recon was doing unnecessary work when it processed a read request to a data block. Unlike metadata reads, Recon does not need to process data reads once they complete, however, it still introduces some latency on the read path by dispatching the completed request to a work queue and acquiring the Recon lock before deciding that it doesn't need further processing. By taking advantage of clearly separated data and metadata, Recon can quickly determine that a read is for a data block before it issues the read request to the underlying device. We optimized this path by checking the data or metadata status of a read request at the time it is being issued, and if it is for a data block, it is returned to the caller immediately upon completion bypassing Recon. This optimization gained us approximately 7% on the read-intensive webserver profile.

	NoFS	Soft Updates	Ext3	RExt3	Btrfs
Consistency Points			X	X	X
Allocation Information		X	X	X	X
Separate Metadata				X	X
Block Identity	X			X	X

Table 5.2: Requirements for Checkable File Systems

5.5 Designing Checkable File Systems

Section 5.3.1 describes the four requirements of a runtime checker that make it feasible to check invariants efficiently at the block layer: 1) well-defined consistency points, 2) easily accessible allocation information, 3) easily distinguishable data versus metadata blocks, and 4) easily available block identity information. In this section, we describe how well various file systems meet these requirements. Table 5.2 provides a summary of our analysis. Then we recommend features that make file systems easily checkable at runtime.

5.5.1 Analysis of File System Design

No-Ordering FS: NoFS [13] aims to provide file system consistency in the face of poorly-behaved hardware that ignores ordering constraints and flush commands. They propose a novel commit-less approach to providing crash consistency by adding a backpointer to every block by using the out-of-band bytes provided by some devices, enabling atomic write of the block and its backpointer together. The backpointer makes it possible to identify the contents of blocks. NoFS performs block allocation based on an in-memory bitmap, thus avoiding any consistency issues between pointers and a persistent bitmap. Determining the allocation status at the block layer is expensive because it requires reading the block and its parent block to determine if a bidirectional pointer relationship exists between them. Unfortunately, NoFS does not provide any ordering guarantees by design, and thus lacks consistency points, and any consistency or location invariants. As a result, it is not possible to check any invariants in NoFS. Bugs in NoFS that cause data corruption would not be easily detectable by an offline checker as well.

FFS with Soft Updates: The soft updates mechanism provides crash consistency in an update-in-place file system without requiring journaling. Soft updates impose a partial order on writes and prevent cyclic dependencies between blocks by using a temporary in-memory rollback mechanism. Blocks and inodes can "leak" after a crash, but this problem is much less severe than blocks or inodes being overwritten while still in use. The ordering of writes allows some invariants to be checked (for example, you can't write a pointer to a newly-allocated block before you initialize the block). However, soft updates are not transactional and thus lack consistency points, and so most file system invariants cannot be checked because data might always be in flight.

Ext3: We have described the Ext3 file system properties in detail in Section 5.3.3. Ext3 provides consistency points and allocation information, but it mixes dynamically allocated metadata (directory data and indirect blocks) with data. In addition, the dynamically allocated metadata blocks cannot be easily interpreted, because they do not contain type information or information about the inode that points to them. We solve this problem partially by using in-memory back references in the checker. Unfortunately, this approach will not protect a metadata block from being overwritten if we have not yet seen a pointer to it. We experimented with a modified version of Ext3 which kept a bitmap for metadata, making it possible to distinguish metadata from data at the block layer without scanning the entire file system for metadata pointers. This required only minimal changes to Ext3 and mkfs, but the design of RExt3, described next, satisfies the prerequisites for online checking of an Ext3-like file system more thoroughly.

RExt3: RExt3 [53] is a variant of Ext3 that is optimized for a fast, offline file system checker called ffsck. Speeding up offline fsck involved two changes to the file system format, the co-location of metadata within metadata regions, and the addition of backpointers associating dynamically allocated metadata with their corresponding inodes. The separation of metadata and data into two regions makes it possible to distinguish between them with low overhead. With the addition of backpointers, the runtime checker for RExt3 will not need to use in-memory back references, thus reducing the memory overhead of the checker.

Btrfs: We have described the Btrfs properties in detail in Section 5.3.4. Btrfs provides consistency points, and it uses a separate extent tree to store allocation information. The extent records specify whether an allocated extent is data or metadata, and also record backpointers for the extent. Since Btrfs allows snapshots, some extents (both data and metadata) may have multiple parent blocks which point to them. A runtime checking system can identify metadata by its placement in a designated area, or by looking up the metadata flag in the extent tree. Furthermore, the contents of a metadata block can be identified based on the header structure shared by all metadata blocks. The shadow paging location invariants are easier to verify than their journaling equivalents because there is less state that needs to be tracked.

5.5.2 Design Recommendations

Based on our analysis of these file systems, we now suggest design features that enable efficient runtime checking of file systems. We expect that these same features will help implementing other file-system aware storage applications, such as differentiated storage services [57]. Consistency points are essential for runtime checking. While new file systems, possibly running on new hardware, may avoid providing consistency points, the resulting loss in protection is a serious issue. Easily accessible allocation information at the block layer, such as in bitmaps in fixed locations, allows enforcing location invariants efficiently. Other applications, like scrubbers and secure delete utilities, can also benefit from knowing the allocation status of a block. Separating data from metadata in well-defined regions allows distinguishing between them with low overhead because there is no need to lookup this information in bitmaps or trees. This approach also allows other policies, such as replication and placement, to be applied to contiguous metadata regions with ease. Fortunately, the mixing of metadata and data for performance reasons has been obsoleted by large disk caches [53]. Finally, backpointer information helps identify blocks at the block layer efficiently. This information is especially useful for dynamically allocated metadata in update-in-place file systems, because the checker may need to interpret an arbitrary block without knowing its position in the file system tree.

5.6 Summary

We have presented the design of runtime file system checkers that can reliably detect file system bugs before they cause file system inconsistency. We show that the runtime checker needs to check location invariants on every write. These invariants enforce the atomicity and durability properties of the file system, helping preserve the integrity of committed transactions. Together with checking consistency properties on commit, the checker can provide the strong guarantee that every block write will preserve file system consistency.

We have implemented runtime checkers for the Ext3 journaling file system and the Btrfs copy-on-write file system. Our experimental results show that while consistency checking imposes some performance overhead, checking location invariants has almost no additional overhead. The Ext3 file system checker has low overhead

but the Btrfs checker has higher overhead due to a higher metadata load. Btrfs keeps a log to enable fast sync operations. Future work could implement our journaling invariants for this log.

We have shown that four file system features ease the design of runtime checkers, and enable checking invariants efficiently: 1) consistency points at which the file system is expected to be consistent on disk, 2) easily accessible allocation information at the block level, 3) distinguishable data versus metadata blocks at the block layer, and 4) backpointers for block typing and identification. We expect that these file system features will benefit other file-system aware storage applications as well.

Chapter 6

Ingot: Consistency Invariants for In-memory Data Structures

In this chapter we present Ingot, a framework for reliably and efficiently detecting consistency violations in byte-addressable persistent memory. Unlike block-storage based systems, applications that use byte-addressable persistent memory can bypass the operating system, and so a different approach than Recon is needed. We also develop the idea of *differential invariants* more fully than in the previous chapters, as we found that during the development of Recon, it was challenging to correctly write the local consistency invariants correctly in an ad-hoc way.

As new persistent memory technologies are being developed, the line between persistent storage and application memory state becomes blurred. New storage and interconnect technologies are driving a shift towards data structures mapped directly into persistent memory [10, 14, 16, 23, 40, 51, 56, 82]. An application can thus bypass the operating system and directly update persistent memory by using load, store and flush instructions. Likewise, RDMA over fast networks is being used to implement storage systems using new models of memory-oriented computation [22, 42, 46]. These approaches decrease latency by bypassing serialization, deserialization, and much of the network stack, but the "wide" interface of arbitrary loads and stores for persistence and communication leaves storage vulnerable to corruption due to logic errors in the application.

Persistent memory applications must take care that they can recover to a consistent state after a crash in case the application is abruptly terminated, whether by power loss or operating system intervention. New libraries and frameworks have been proposed to provide crash-consistent abstractions for application programming to make this easier. We observe that these crash consistency mechanisms for persistent memory supply the prerequisites for run-time consistency checking, similar to crash consistency for block storage. Once an application has been modified to use persistent memory in a crash-consistent manner, run-time consistency checking requires very little modification to the application.

Our aim is to help developers catch corruption-causing bugs in persistent memory applications, rather than for applications in general, because we leverage crash consistency for invariant checking. The same techniques could also be applied to data in volatile memory, if applications are modified to use the same frameworks for consistency, but the performance tradeoffs may be less acceptable than in persistent memory where it is necessary. Furthermore, persistent corruption may be more damaging than volatile corruption, and more difficult to recover from.

Ingot is able to catch corruptions that violate the consistency of persistent memory data structures. The main

challenge with checking consistency invariants at runtime is that the checking effort grows with the size of the data structures. Consider this simple consistency invariant for a hash table: every key should be hashed into the correct bucket. This consistency invariant can be checked by iterating over every element in the table and ensuring that its key matches its bucket. This check is linear in the size of the table, making it expensive to perform frequently, especially for large persistent hash tables. However, run time checking needs to be scalable for corruption to be detected as soon as possible. This requirement is especially important for languages with pointer manipulation, where a bug can corrupt an arbitrary structure.

Ingot scales consistency checking by using *differential invariants*, whose checking time is proportional to the amount data accessed by the application, rather than the data structure size. A differential invariant is a type of incremental check that uses knowledge of both the "old" and the "new" data structure states to delineate the subset of state that must be checked after an update. It minimizes the amount of checking by ensuring that the updated state *remains* consistent, given that it is *initially* consistent.

Ingot is designed for unmanaged languages (i.e., languages with pointer manipulation like C or C++) and thus needs to check arbitrary stores that may corrupt any data structure. In order to do this reliably, it is integrated with Atlas, a persistent memory framework which uses binary instrumentation to log all persistent writes. Ingot interprets these writes in terms of program data structures using information supplied by the application programmer, and determines what differential invariants need to be checked before the writes are allowed to become durable. A user of Ingot needs to follow three steps: first, annotate persistent allocations with type information; second, supply data structure specific functions to help Ingot convert between addresses and field names, and third, write the differential invariants that Ingot is supposed to check.

Differential invariants in Ingot consist of a trigger, which is a structure or a field of a structure that may change, and a check, which is the set of properties to be checked when the trigger field is changed. For example, updating a pointer in a hash table bucket could trigger an invariant that checks the hash of the element that the bucket points to.

To simplify writing differential invariants, we observe that many consistency invariants can be expressed in relational calculus, a declarative first-order logic language for formulating database queries [15]. Using relational calculus helps simplify reasoning about differential invariants, because it allows using techniques designed for materialized view maintenance [31, 35] and differential relational calculus [61]. We describe methods for *differentiating* a consistency invariant with respect to the underlying data structures, and then converting the resulting (declarative) differential invariants into efficient low-level assertions. We also show how to convert invariants that do not fit within first-order logic, such as invariants involving aggregation and recursion, into their differential versions.

These invariants must only be checked when the data structures are expected to be consistent. Checking intermediate states, such as when a data structure is being updated, possibly concurrently, would raise false alarms and additional failures (e.g., crashes). Ingot reuses the transaction logging framework to carefully construct consistency points that represent boundaries between completed operations. Data structures are expected to be consistent at these points, allowing invariants to be checked correctly.

To evaluate our approach, we show how Ingot can be used to protect a persistent red-black tree, with complex recursive consistency invariants. We also modified the Redis key-value store for persistent memory, and enforced a variety of consistency invariants on its internal data structures using Ingot. Our evaluation shows that differential invariants can catch randomly injected corruptions that affect data structure consistency. We also show that differential invariants can be checked much more efficiently than the corresponding consistency invariants. Finally, we evaluate Ingot in a multithreaded environment by using a variant of Thredis [79], an experimental, multithreaded

version of Redis.

Our main contribution is to show that differential invariants provide a powerful and robust method for ensuring consistency of persistent memory state in the face of various types of random corruptions and application bugs. To showcase our approach, we have designed the Ingot framework for ensuring the consistency of unmanaged code, unlike previous work that targets Java [1, 66, 71].

To handle arbitrary, possibly-erronenous stores, it uses the transaction logging framework of persistent memory applications to intercept stores, interprets them in terms of program data structures, and then generates old and new data values needed for the differential invariants. Furthermore, this framework is unique is being able to ensure consistency in the presence of concurrent updates. It handles concurrent updates by triggering the differential invariants on the changed data structures at inferred consistency points in the program where data structures are expected to be consistent.

The rest of the chapter describes and evaluates Ingot. Section 6.1 describes our assumptions and fault model. Section 6.2 describes how consistency invariants are systematically converted to differential invariants. Then Section 6.4 describes our implementation of differential invariants in the Ingot framework, and Section 6.3 discusses advantages and limitations. In Section 6.5, we describe our evaluation results.

6.1 Programming and Fault Models

This section outlines the assumptions made by the Ingot system about the application and hardware which define the scope of what Ingot is able to check.

6.1.1 Adapting Applications for Persistent Memory

A program designed for disk-based persistence needs modification to take full advantage of byte-addressable persistent memory. Rather than manipulating data in volatile memory and then serializing to / deserializing from a file, the application can allocate its structures in *persistent memory* and manipulate them directly. This creates two categories of objects, persistent objects (allocated on the persistent heap) and volatile objects (allocated on the ordinary heap). After a restart of the application, the persistent objects will remain at their previous addresses, while the volatile objects will not.

In order to ensure that allocations and updates of persistent objects are durable and atomic, it is helpful to use an existing library or framework to access persistent memory. In this implementation of Ingot, we assume that the application is using the Atlas framework [10], which was created to minimize the effort of adapting existing programs for persistent memory. Atlas assumes that the program is written using lock-based synchronization. To use Atlas, the programmer specifies which memory allocations should come from the persistent heap by calling `pmalloc` instead of `malloc`. Optionally, the programmer also specifies the begin and end of atomic operations if they do not correspond to mutex lock and unlock pairs. Otherwise, Atlas infers atomic operations by instrumenting and analyzing the state of mutexes at run time, relying on the assumption that the program is free of data races. Atlas ensures that all writes to the persistent heap are made atomic by instrumenting heap stores, filtering out those destined for the volatile heap, and using undo logging to create a crash-consistent record of the writes destined for the persistent heap.

6.1.2 Programming Model

Ingot is designed for applications written in low-level, unmanaged languages such as C. We target applications written in C for two reasons. First, they are more susceptible to corruption due to pointer errors. Second, they allow us to evaluate the overhead of Ingot in a "worst case" fashion. The program has no language run-time overhead, and does not provide any run-time type information, and thus Ingot requires additional work to safely derive such information.

Our implementation targets existing applications that use conventional, lock-based synchronization for concurrency control. We adapt these programs for persistent memory, and ensure crash consistency by using the Atlas framework. However, as described in Section 6.3, the Ingot approach is also applicable to other programming models, such as transactional memory or shadow paging.

6.1.3 Fault Model

Ingot provides a strong fault model that protects the persistent heap of an application against corrupting writes, without relying on the programmer to invoke or use the fault detection API correctly. The corrupting writes may be caused by logic errors, concurrency errors, pointer errors, or CPU errors.

Ingot can only protect against writes that originate in the application code. Our implementation relies on instrumenting stores to the persistent area (see Section 6.4), and so we cannot protect against corrupt writes by the operating system, or writes originating in self-modifying code and uninstrumented libraries. Concurrency or CPU errors that corrupt the destination of a store between logging and execution of the store could possibly also escape detection. As an alternative, a hardware monitoring system that is able to snoop on cache traffic would enable checking all data structure changes more robustly. We also assume that the memory subsystem detects hardware memory errors reliably (e.g., using SECDED ECC RAM).

We assume that the Ingot framework has no bugs. While this is unlikely, we ensure that Ingot and the invariants do not perform any persistent writes. A bug in the framework or invariants may cause a false positive, but bugs in the framework will only cause a *miss* of corrupt write due to a bug in the application if the two bugs coincided.

6.2 Differential Invariants

This section describes differential invariants, a novel approach for checking consistency invariants efficiently. Differential invariants compute invariants incrementally based on knowledge of what has changed since the last check, thus avoiding unnecessary work in the face of a small set of updates to some large state.

Next, we explain how we systematically derive and implement differential invariants from consistency invariants. Then we describe how aggregation and recursive invariants can be supported. Finally, we discuss the benefits of differential invariants over existing approaches.

6.2.1 Deriving Differential Invariants

We describe the process of converting consistency invariants into their differential versions using a linked list example. Suppose an application uses a sorted linked list, and we wish to enforce the invariant that the list is in sorted order. This should remain true after each atomic operation, so it is a consistency invariant.[1] Checking this invariant requires each node of the list to be checked, making it expensive for a large list. In a language like C,

[1] A linked list has additional invariants to guard against list truncation, cycles, etc., but we avoid discussing them for simplicity.

```
struct Node { Node *next; int value; }

∀ Node n: (n.next == 0)
      || (n.value <= n.next->value)
```

Figure 6.1: Sorted linked list and its consistency invariant

the list may be erroneously overwritten through a stray pointer in some code that shouldn't be modifying lists at all. Since we don't know whether this list may have changed, we need to perform the check after each operation.

To convert a consistency invariant into a differential invariant, we start by expressing the consistency invariant in relational calculus [15], as shown in Figure 6.1. The invariant consists of first-order predicates quantified over the objects and their fields in the application's heap. This invariant is quantified over a single variable, but it also involves a pointer dereference that introduces a second `Node` object into the relationship. In relational terms, this represents a *join* between the first node and the pointed-to node. To express this join, we replace the pointer dereference by introducing another `Node` variable `m`, and an implication, to obtain an *ordering* invariant:

$$\begin{aligned} &\forall\ \texttt{Node}\ \ n,m : (n.next = 0) \vee \\ &\qquad ((n.next = m) \Rightarrow (n.value \leq m.value)) \end{aligned} \tag{6.1}$$

We also need to ensure that the *next* field points to an object of the correct type, which leads to the following *type-safety* invariant. Here, Node(x) is a predicate that asserts that address x is a Node.

$$\forall\ \texttt{Node}\ \ n : (n.next = 0) \vee \texttt{Node}(n.next) \tag{6.2}$$

Differential invariants are obtained by applying *differential relational calculus* [61] to the invariants shown in Equations 6.1 and 6.2, with respect to the underlying variables (`n.next`, `n.value`, and `m.value`). The resulting differential invariants are triggered by changes to these variables. The ordering and type-safety differential invariants evaluate the same checks as shown in Equations 6.1 and 6.2, but without requiring the "forall" clause on the changed variable, since it gets bound. For example, when `n.next` changes, we need to check both type-safety and ordering, since this variable occurs in both invariants, and the corresponding differential check is:

$$\begin{aligned} &update(n.next) \Rightarrow (n.next = 0) \vee \\ &(\texttt{Node}(n.next) \wedge \\ &\quad \forall\ \texttt{Node}\ \ m : (n.next = m) \Rightarrow (n.value \leq m.value)) \end{aligned} \tag{6.3}$$

Unlike the consistency invariant shown in Figure 6.1, which must check every node of the list, the differential check in Equation 6.3 only needs to check updated nodes. We discuss changes to the `value` field in the next section.

6.2.2 Implementing Differential Invariants

Differential invariants in Ingot consist of a *trigger*, which is a structure or a field of a structure that may change, or an object that is created or destroyed, and a *check*, which is the set of properties to be checked when a trigger fires.

```
process_diff_Node(Node x, field_id f) {
  switch(f) {
  case NODE_NEXT:
    if(x.next != 0) {
      assert(get_type(x.next) == TYPE_NODE);
      assert(x.value <= x.next->value);
    }
  case NODE_VALUE:
    if(x.next != 0) {
      assert(get_type(x.next) == TYPE_NODE);
      assert(x.value <= x.next->value);
    }
    Node * n = get_parent(x);
    assert(n == 0 ||
           (n->next == x && n.value <= x.value));
  case NODE_CREATE:
    /* ... same as NODE_VALUE... */
  case NODE_DELETE:
    Node * n = old(get_parent(x));
    assert(get_type(n) != TYPE_NODE ||
           n.next != x);
  }
}
```

Figure 6.2: Differential invariant for the sorted linked list

Figure 6.2 shows the implementation of the differential invariant for the sorted linked list. Each case in the switch statement represents a trigger, and the corresponding check, written as a series of assertions. When the `next` field changes, the check is as shown in Equation 6.3. To check type safety, we use Ingot's `get_type()` operator that returns an enumerated value representing the type of the object at that address, or 0 if the memory is unallocated.

When the `value` field changes, it may either be `n.value` or `m.value` in Equation 6.1. Differential invariants require we check both cases. With `n.value`, we need to compare its value with the next node. With `m.value`, we need to find the `Node n` that points to `Node m`, i.e., the parent of `Node m`.

To locate the parent efficiently, we need to track the parent pointer of each node. Since this situation is common for many data structures, Ingot allows the programmer to attach auxiliary fields to any allocated object. The programmer explicitly sets and clears these fields as they update their data structures, and invariant checks can access these fields. In Figure 6.2, the `get_parent()` function returns the auxiliary parent pointer field. Consistency invariants can also be specified to ensure the integrity of these fields. For example, when a `next` pointer is set or cleared, they would ensure that the parent pointer is set or cleared accordingly.

When an object is created, we check the new values of its fields. When an object is deleted, we perform all checks that access the old values of its fields. Other invariants may be triggered specifically by object creation and destruction. For example, when a node is deleted in Figure 6.2, we check the type-safety invariant on the node's former parent by using Ingot's `old()` function to access the parent (if it exists) and check that it does not have a dangling pointer to the deleted node. Without access to this *old* value, we could not efficiently find the *potential* dangling pointer to check.

6.2.3 Aggregation

A simple extension to relational calculus includes *aggregation* operators, such as SUM and COUNT. Invariants using these operators often occur in resource-tracking or summary statistics. For example, if we are using refer-

ence counting to determine when to free an object, we want to know that the total number of pointers to an object is reflected in the reference count.

Differential invariants using such operators can be computed using finite differencing [31]. For the reference counting example, we count the references that have appeared (by tracking newly created pointers to an object), and subtract the references that have disappeared (i.e., by tracking the old value of overwritten pointers), and compare that to the difference between the new and old values of the object's reference count.

6.2.4 Recursion

Many invariants are expressed in recursive form, especially invariants on graph structures. For instance, suppose our program has multiple linked lists: some are ordered ascending, some are ordered descending, some are unordered, and the invariant to apply depends on the list head at the beginning of the list. In this case, we need to determine if a node is a member of a particular list. The membership property is *recursive*: a Node B is a member of list A iff A points to B, or Node C points to B and C is a member of list A. These recursive and transitive closure properties cannot be directly expressed in relational calculus [2].

A recursive invariant may relate objects that are arbitrarily distant from one another, and it may be violated due to updates to intermediate objects. This makes it hard to write checks and reason about the correctness of these checks, especially in the face of multiple updates. Moreover, it is hard to ensure that differentially checking a recursive invariant takes time proportional to work done by the application.

We replace recursive invariants by *memoizing* the values they would have computed, and storing the memoized values as auxiliary fields in objects. This method is similar to storing the parent of a node for the non-recursive, ordering invariant, described in Section 6.2.2. For example, an auxiliary field can maintain the head of the list at each node, making the list membership check a non-recursive (or local) predicate. We require this auxiliary field to be updated by the programmer when updating a node, and additional invariants can be used to ensure their integrity.

We illustrate this approach of "flattening" recursive invariants by showing how it can be applied to differentially check the integrity of a more complex data structure, a red-black tree. A red-black tree is a binary search tree that ensures that every leaf node is at most two times the depth of any other leaf node. Nodes are colored "red" or "black", which affects rebalancing operations as nodes are inserted or deleted. A red-black tree has three invariants: 1) alternation: a red node only has black children, 2) black-depth: the number of black nodes along any path from a node to any of its leaves is the same, 3) ordering: the tree is ordered. The red-black alternation invariant is a local invariant that is easily written as a first-order predicate, while the other two are recursive invariants.

The black-depth invariant can be memoized simply by maintaining the depth count as an auxiliary field at each node. The differential check can then ensure that a node's depth count is correct by checking it against the depth count of its left (or right) child and of its parent. Updating the depth count is efficient because many rebalancing operations in the red-black tree do not need a traverse all the way to the leaf to determine whether to perform a rotation and/or re-color a node. Thus we only need to adjust the black depths of the immediate neighborhood, and occasionally the ancestors as the rebalancing operation propagates upwards.

The ordering invariant at a node requires checking consistency with the node's children by recursively looking up its predecessor (i.e., the right-most leaf of the left child of the node) and its successor, and comparing the keys. In addition, the node's ancestors need to be checked to ensure that the node lies in the correct part of the tree. However, these checks take $O(log\ n)$ time. Instead, we memoize a lower and an upper bound value at each node, so all keys in the sub-tree rooted at this node fall within this range. The differential check then ensures that a left

child's lower bound is greater than or equal to the node's lower bound, and the left child's upper bound is less than or equal to the node's key. Updating these bounds is a little more expensive. For example, when a node is inserted, it may need to adjust these bounds for all its ancestor nodes. However, node deletion does not require updating any bounds. In all cases, we find that updates to auxiliary fields fit within the existing tree traversal, ensuring that checking overheads are proportional to the amount of data processed by the operation.

6.3 Discussion

Previous work on heap assertions focuses on checking a snapshot of the heap [1, 66, 71]. The ability to access the *old* values of variables is the key difference between *differential* invariants and these approaches, which only have access to the current (updated) state.

For example, the deletion trigger in Figure 6.2 checks the type-safety invariant on the `next` pointer of a `Node` in the ordered linked list. The check ensures that if there *was* a `Node` that *used to* point to the deleted object, either it has also been deleted, or its `next` pointer has been changed. This is a specific instance of *foreign-key consistency*, where we must ensure that a foreign key refers to a valid object.

The ability to access the old state is also necessary for testing whether the value of a predicate on some field(s) has changed. For example, this can be used when an invariant dictates that a particular property is held by one, and only one, object. When a relevant field is updated, we can determine whether this was previously the "special" object, and if this is no longer so, we must check that some other object has acquired the property in this transaction. Likewise, if an object gains the property, we must ensure that we also saw another object lose this property.

The availability of old and new values makes it possible to efficiently check the reference count invariant differentially, as explained in Section 6.2.3. For comparison, we show how this invariant could be checked with DITTO [71], which uses function-level memoization to incrementally check invariants that are written as recursive, side-effect-free functions in Java. Dependencies between function invocations and the data used by these invocations are tracked to help determine which functions need to be recomputed when their input data is modified, with memoization helping limit recomputation. This approach is powerful (in terms of the invariants that can be expressed), but it can still require significant recomputation, and it incurs high space costs, proportional to the number of recursive function invocations, and the size of all their dependencies.

The straightforward approach in DITTO for checking the reference count invariant is to walk the tree looking for object pointers, and then for each object found, walk the tree again accumulating a reference count which is checked on return to the top-level walk. This method requires $O(n^2)$ function invocations, for a tree with n nodes (and consequently, $O(n^2)$ space for memoization, since the parameters have constant size). When the pointers and reference counts are updated, changes propagate up the tree of function invocations representing the traversal of the data structure, until they meet and are reconciled. Assuming a fairly balanced tree of objects, we can expect the computation time to take $O(log(n))$ steps in this method.

Rather than traverse the tree $O(n)$ times, we could pass around a data structure that keeps track of the number of references that have been seen, and the reference count of every object as we walk the tree. Afterwards, we can iterate over this data and check that the observed refcount matches the refcount in the object. This approach still requires $O(n)$ function invocations, but to make memoization work, we can't pass around a reference to a mutable data structure – we must pass it by *value*. The size of the parameters for each invocation will be $O(n)$, so once again the space overhead is $O(n^2)$. The per-invocation overhead could possibly be reduced by using an optimized immutable map (e.g., using copy-on-write), but memoization still has other costs such as equality testing between

`ingot_alloc(size, type, isArray)`	Allocates persistent memory and associates it with a type, which is an application-defined constant.
`set_parent(object_addr, value)`	Allows the programmer to store a back pointer (or other auxiliary field) using Ingot's metadata store.
`get_parent(object_addr)`	Returns the back pointer (used during invariant checking)
`field_id offset_to_field(type, offset)`	Identifies the field at a given offset within a given type.
`size_t size_of_field(type, field_id)`	Returns the size of a given field in a given type.
`process_diff(trigger, object_addr, type, field_id, new_val, old_val)`	Implements the differential invariant checks; `trigger` is one of CREATE, DELETE, or UPDATE.
`check_deferred_invariants()`	Called once all log processing is complete to finalize checking (e.g., for invariants involving aggregates).

Table 6.1: Ingot API functions.

entire maps.

In this case, Ingot has a significant advantage over DITTO's incremental checking, because we use both the old and new values of the modified pointers to determine which reference counts should have been updated. Instead of memoizing a traversal of the entire heap, we can directly compute the finite difference of the old and new reference counts and compare it with the difference between the new and old pointers to the refcounted object.

By limiting the checks to those expressible in relational calculus, Ingot requires less state and can run the checks more efficiently because differential invariants statically remove the subcomputations that would not be affected by the changes. However, programmers must maintain auxiliary fields for recursive invariants and to speed up checking.

While our implementation of Ingot depends on ATLAS' undo log in order to detect changes and access old and new values of fields, Ingot can be adapted to any system for crash consistency which provides access to the old and new states as well as a commit point. For instance, immediately before a copy-on-write system commits, the new version of data has been written to a new location, and the old data has not yet been freed or overwritten. By traversing the old and new subtrees, Ingot can construct a set of change records, similar to how the Recon system processed changes in the copy-on-write Btrfs file system. Likewise, software transactional memory systems must be able to abort the changes made by the current transaction, and so they must store enough information to construct both the old and the new states immediately before the commit point.

6.4 Implementation

This section describes the implementation of differential invariants in Ingot. We start by describing the Ingot API. Next we explain when and how invariants are checked.

6.4.1 Ingot API

Ingot provides the API shown in Table 6.1. The first set of functions are provided by Ingot to allow an application programmer to allocate persistent memory, associate it with a type, and set or retrieve an auxiliary field using Ingot's metadata store. Type information is used to interpret the writes an application makes in terms of logical structures and fields. The auxiliary field is invariant-specific and used for identifying the parent of an object, membership within a set, etc. The second set of functions are implemented by the application programmer to help Ingot determine which field was modified by a write. For simple data structures, these functions could be generated by macros or an annotation language. The final set of functions are implemented by the

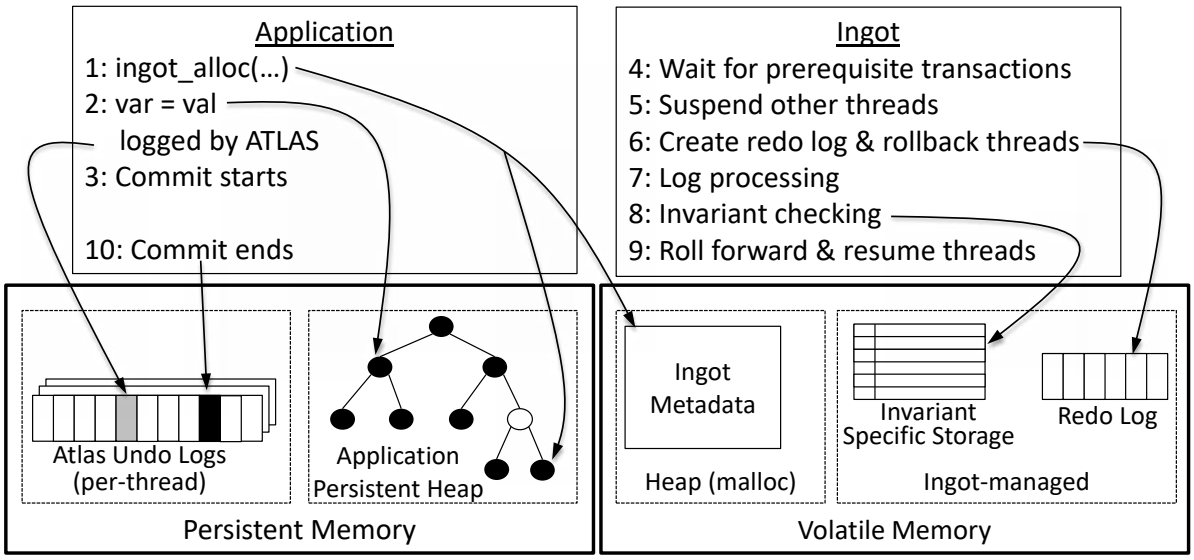


Figure 6.3: Ingot data structures and workflow

programmer to check differential invariants. For each field modified, Ingot calls `process_diff` with all the information necessary to trigger a differential invariant check. After transaction processing is completed, Ingot calls `check_deferred_invariants`.

We now explain how Ingot establishes a consistency point at which invariants can be checked, and how invariant checking is performed. Figure 6.3 provides an overview of this process.

6.4.2 Establishing Consistency Points

To check invariants, Ingot must find a *consistency point* – a point in time when all data structures are expected to be consistent. Programs using persistent memory must ensure that their data structures remain consistent after a crash failure. We exploit this *crash consistency* requirement to find consistency points, which occur immediately before an atomic group of writes becomes durable, and exclude non-durable writes outside of this group.

Ingot supports existing, conventional lock-based applications that are being incrementally adapted for persistent memory, rather than requiring applications to adopt a new programming paradigm [14, 82]. To do so, we use the Atlas framework [10], which provides crash consistency for lock-based programs. Atlas dynamically infers *failure-atomic sections* (which we call transactions) from a program's locks. It ensures that each transaction becomes durable (commits) atomically, and only after every other transaction upon which it depends has become durable. To determine the durability order, Atlas tracks *happens-before* dependencies between writes by instrumenting and logging synchronization operations. Atlas also provides an interface to allocate persistent objects, which we extend for Ingot to track additional object metadata, such as the type (Step 1 in Figure 6.3). Last, Atlas instruments stores to the persistent heap to maintain a per-thread undo log that enables rollback for crash recovery (Step 2 in Figure 6.3).

Ingot checks the changes made by a transaction when a thread reaches a commit point (Step 3). If a committing transaction depends on an uncommitted transaction in another thread, Ingot pauses the committing thread until the prerequisite transactions have reached their commit points (Step 4). Once all dependencies, including cyclically dependent transactions [10], have reached their commit points, Ingot checks the group of committing transactions together as a single transaction (group commit), which is safe because consistency invariants must also hold over a group of committing transactions.

The use of undo logging in Atlas adds a complication – uncommitted writes from other concurrent threads are immediately visible in the persistent data structures, leaving them in a potentially inconsistent state when a transaction begins its commit. During invariant checking we need to ensure a consistent view of the entire persistent state, thus we need to rollback uncommitted updates from other threads. Ingot suspends all other threads (Step 5), and uses their undo logs to temporarily revert uncommitted writes, simultaneously creating a *redo* log (Step 6) to restore these changes after checking is complete. After this rollback, a consistency point is reached.

Ingot then processes the undo logs for all of the committing threads to identify the changes that trigger invariants (Step 7) and checks these invariants (Step 8). Then, Ingot rolls the suspended threads forward using the redo log and resumes them (Step 9). The commit ends by making the checked updates durable (Step 10). The durability and crash consistency provided by Atlas ensures that the last state checked by Ingot will be recovered after a crash.

Atlas assumes that the persistent memory programs are data-race free. With data races, Atlas' durability ordering may be incorrect; for Ingot, a write-write data race may cause an incorrect rollback in Step 6 in Figure 6.3. Consider transactions T1 and T2 that race to update X, T1 overwrites T2's correct update with an incorrect value, and T1 commits before T2. While checking T1, T2 is rolled back, which restores the orginal (valid) value of X and saves the current value (T1's incorrect update) in the Redo log. This rollback may hide an invariant violation caused by T1, but the roll-forward of T2 would restore T1's update and Ingot would detect the violation when T2 is checked.

6.4.3 Checking Differential Invariants

We first describe Ingot's data structures, and then the implementation of the triggers and the differential checks.

Ingot Data Structures Ingot interposes on the persistent memory allocation and deallocation routines supplied by Atlas, and maintains an *allocation map* that tracks information about objects allocated on the persistent heap (Step 1 in Figure 6.3). The map key is the base address of the object, and the value includes an encoding of the object type, its size, status flags, and the auxiliary pointer (e.g., a back pointer or set membership).

The Ingot allocation map is kept thread-safe by keeping a thread-local buffer of updates until it is time to commit. The thread processing a group commit is responsible for executing the buffered updates of each committing thread on the main metadata structure.

Ingot also tracks the set of objects that were created or destroyed within an executing transaction. When an object is allocated, we mark the 'new' flag in its allocation metadata. Freed objects may still be needed by invariants when the transaction commits, so we mark the 'deleted' flag in the allocation metadata of these objects, but defer deallocation until checking completes. Deferred deallocation (and hence reallocation) ensures that memory locations cannot change types within a transaction, which simplifies processing of the undo log. It also makes it possible to assert that if an object was just created, there were no pointers to it in its previous state.

When a given thread reaches a commit point, Ingot must efficiently distinguish between threads it can roll back and threads it must wait for in order to satisfy happens-before dependencies. If we have acquired a lock from a thread during an uncommitted epoch, we must wait for that thread to join us at commit. Ingot tracks this by keeping a per-thread record of the most recent epoch numbers of each other thread that a given thread depends on.

Triggers and Checks There are three types of data structure change events that trigger differential invariants: object creation, object deletion, and field updates. The new and delete flags are used to trigger the appropriate invariants for newly created and deleted objects. Ingot keeps a list of such objects so that it does not have to search

its entire metadata collection.

Ingot detects field updates by parsing the Atlas undo log (Step 7 in Figure 6.3). We can reliably detect all application updates because Atlas maintains the undo log by instrumenting the application's stores, and so a bug in the application will not cause a check to be bypassed.

Ingot uses the undo log to generate the old and new values of the updated fields of an object, which we call *diffing*. Diffing interprets writes in the undo log by looking up the address of the write in the allocation map. This map is implemented as a Judy array that simulates content-addressable memory efficiently, returning the object (if any) on which the write was performed [73]. Next, diffing uses the object's type and the offset of the write to determine the field that was modified, using the `offset_to_field` and `size_of_field` functions. The checks are provided with the trigger, the object, its type, a field identifier, the old value of the field, and the new value of the field that triggered the invariant, as shown in Table 6.1.

Invariant checks must be provided with the oldest value of a field in the committing thread's log because this value is part of the (checked and consistent) pre-transaction state. To find the oldest value, log entries are traversed from oldest to newest; when an address is first seen by the diffing process it is marked and newer log entries for that address are ignored.

Since memory freed by a transaction cannot be reallocated until after the transaction commits, an object that has both the new and the delete flags marked must have been created and destroyed during the scope of this transaction. Such objects cannot have any impact on the pre- or post-transaction consistency, so we skip diffing them.

While differential invariants are being checked, they may need to temporarily materialize a relation between multiple different updates. For example, for the reference count invariant discussed in Section 6.2.3, we use a collection to count the number of references that have appeared and disappeared for each object during diff processing. At the end of the processing, we check the reference count invariant for all objects in the collection (Step 9 in Figure 6.3). These collections are destroyed once the thread has been checked.

Suspending and Resuming Threads

As Linux lacks kernel support for suspending specific application threads, Ingot uses signal handlers and user-defined signals to temporarily suspend the threads being rolled back. Because these threads may have been suspended while holding locks (e.g., within malloc or printf in libc), the invariant checking code that executes while other threads are suspended must be written as if it was within a signal handler. To facilitate dynamic allocation for invariant checking, Ingot provides a bump allocator to allocate temporary memory during this interval. This allocator is used for both the redo log, as well as any invariant-specific structures that need to be allocated. For example, a reference-count tracking invariant might use this space to accumulate counts of pointers that have been set or cleared during the transaction.

When an invariant is violated, it is safe to restart the program, which preserves the durability of all committed transactions. An alternative is to transparently restart non-corrupting uncommitted transactions, similar to Membrane [77]

6.5 Evaluation

We evaluate Ingot by applying it to a red-black tree, and to Redis, an in-memory, key-value store. We adapted both for persistent memory by using the Atlas framework and designating certain allocations as persistent. In each case, we first assess how well differential invariants can detect various types of random corruptions, simulating the behaviour of broad classes of bugs. Then we present the performance impact of using differential invariant

checking, and compare against global consistency checking.

6.5.1 Experimental Setup

All experiments were run on Intel Xeon E6-2650 2.0Ghz 16-core machines with 16GB RAM. ATLAS uses pre-allocated RAM to emulate persistent memory. While we do not introduce additional persistent memory write latency, we do enable all the necessary ordering and flush operations to ensure crash consistency. Persistent memory is expected to have higher write latency than DRAM, which will generally diminish Ingot's relative overhead for crash-consistent applications, although it will increase the overall spread between a volatile and persistent version of an application.

For the corruption experiments, we perform three different types of random corruptions: stray writes, corrupt writes, and missed writes.

Stray write: we simulate a thread performing a stray write to some object by corrupting a random heap-allocated object at a random offset, and logging the corrupting write. This type of corruption is often the result of a pointer-related error, such as a dangling pointer or a buffer overflow.

Corrupt Write: we simulate a logic error in which a field is incorrectly modified by picking a log entry at random and corrupting the new value by incrementing or decrementing it. This type of corruption can also happen as a result of race conditions, where a stale value accidentally overwrites a logically-newer value due to incorrect synchronization.

Missed write: we simulate a logic error in which an update is missed by picking a log entry at random and reverting the change made by the logged update.

We run the corruptor 500 times for each type of corruption. Each run injects a single corruption and then waits for either an invariant violation (*Caught*), a program crash (*Crash*), or for the experiment to run for another 1000 operations. In the last case, we manually analyze the corruption and classify it as *Safe* (i.e., consistency invariant is not violated, but will not cause a crash), or *Latent* (i.e., consistency invariant is not violated, but may cause a crash).

6.5.2 Red-Black Tree

We implemented a persistent red-black tree in user space, based on the augmented red-black tree used in the Linux kernel, and added invariant checking to it. This implementation stores a key, a value, and an augmented field that tracks an associative computation over all the children of a node. In our implementation this augmented field holds the maximum of the value fields in a node and its children. In addition to the color alternation, black-depth, and ordering invariants on red-black trees, as described in Section 6.2.4, we added three other invariants to ensure: 1) pointer type-safety, 2) consistency of the double link between parent and child, and 3) consistency of the augmented field. While the augmented field is recursively defined, its value is already memoized, and so it can be checked using a local predicate.

Figure 6.4 shows our corruption results for the Ingot-protected red-black tree. All corruptions were caught by the invariants or were considered safe. The safe corruptions were of two types: corruption of the randomly generated key fields of nodes during initialization (which affects where nodes are inserted, but not the consistency of the red-black tree), and corruption of the upper and lower bound fields. These bounds are not tight (which improves update performance), and so some corruptions to these bounds do not cause errors.

Figure 6.5 compares the performance of the red-black tree with full differential invariant checking to three other configurations: a baseline with no auxiliary fields (just ATLAS persistence), the baseline + auxiliary field

Corruption Type	Results			
	Caught	Safe	Latent	Crashed
Stray Write	464	46	0	0
Corrupt Write	454	46	0	0
Missed Write	471	29	0	0

Figure 6.4: Red-black tree corruption results

Number of nodes	100	1000	10000
baseline (ATLAS)	1.01	10.39	113.1
baseline + auxiliary fields	1.97	22.59	257.3
Ingot (differential) checking	3.05	36.01	418.8
Global checking	1.32	40.47	5954.1

Figure 6.5: Red-black tree performance results

updates necessary for differential checking of recursive invariants, and a version that performs the global consistency check built in to the Linux rbtree-test program at every operation. The global check enforces most of the invariants that we do but does not provide pointer type safety.

The test program inserts and then deletes a specified number of nodes. We run the experiment for three different tree sizes: 100 nodes, 1000 nodes, and 10000 nodes. Figure 6.5 shows the run time for each experiment in millions of cycles, averaged over 5 runs. With 100 nodes, the global (non-differential) check is more efficient than the differential version. With 1000 nodes, the differential approach catches up, and with 10000 nodes, differential checking is over 14x faster, while providing stronger guarantees (i.e., safety from pointer corruption).

These red-black tree experiments are a micro-benchmark in which the application is only performing data structure update operations. In this case, Figure 6.5 shows that our overheads lie between 300-370%. These overheads arise because the Ingot invariants provide strong type-safety guarantees and check several complex consistency properties. We also see that a significant proportion (200-230%) of the checking overhead is due to updating the auxiliary fields. Currently, these fields are maintained in persistent memory, which adds to the cost of logging and flushing updates to these fields consistently. These fields could also be stored in volatile memory, as they can be easily recomputed after a crash.

6.5.3 Redis

The Redis key-value store supports widely used data structures, and is commonly used for production-level applications. Redis provides logging or snapshot based persistence to disk. By augmenting Redis with the ability to use persistent memory, it becomes a low latency, synchronously persistent store, and a good platform on which to evaluate Ingot. Key-value stores like Redis are among the broad family of NoSQL applications used to maintain web application state.

To test Ingot in a multi-threaded environment, we also adapted the *Thredis* project [79], which adds the ability to dispatch commands to multiple worker threads as well as the necessary synchronization. Thredis was designed to offload processing of long-running commands. We modified it to dispatch short SETs and GETs as well. This was done as a proof of concept and a stress-test for Ingot; it is not a recommended architecture for applications that serve many short requests (like Redis GETs and SETs) on a shared dictionary.

Instrumenting Redis

We instrumented Redis for persistent memory using Atlas by distinguishing between persistent and non-persistent allocations. Our current persistent Redis prototype supports operations on string/integer values, while Redis also

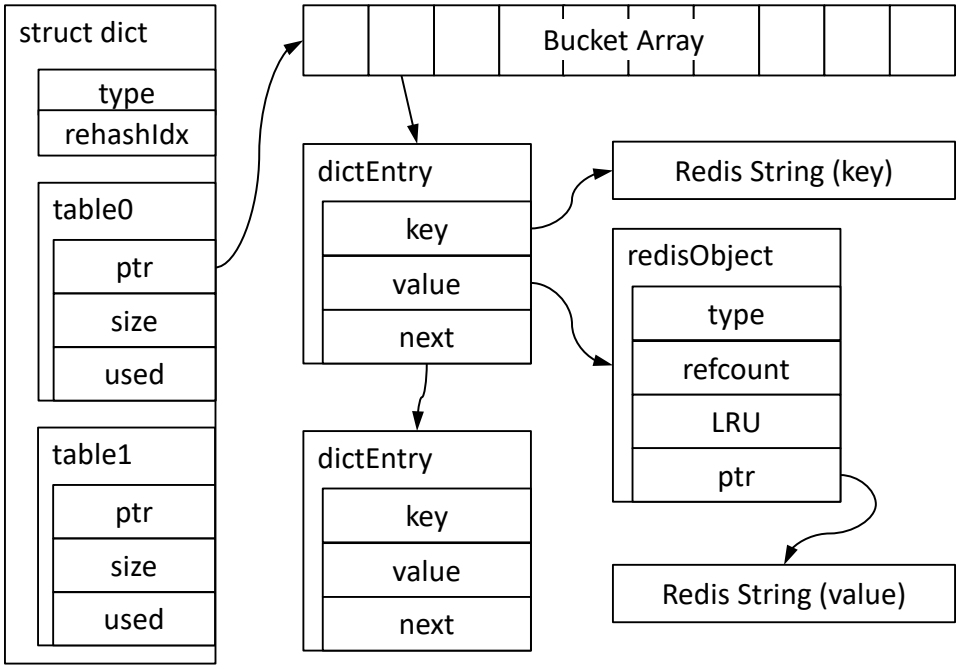


Figure 6.6: Redis data structures

supports other types, such as lists and sets. Our changes to Redis required 312 lines of code, including persistent region initialization, type metadata for Ingot (17 lines), and distinguishing between persistent and non-persistent objects in generic code.

Redis is single-threaded and thus has no locks from which to infer transactions. We establish consistency points by inserting calls to begin and end a transaction in the main command processing loop. Thredis, by contrast, has several threads and many locks, so invariant checking is triggered whenever a thread releases all of its locks.

Redis Data Structures and Invariants

This section describes the Redis data structures we protect, and the consistency invariants we use to protect them. While our invariants are specific to Redis' data structures, they are representative of the internals of many in-memory storage applications that use dictionaries, linked lists, reference-counted objects, and occasionally complex pointer relationships. For example, a pointer to a Redis string doesn't point to the beginning of Redis' string data structure, but to the first character of the string, which is preceded by a header.

The principal data structure in Redis is a dictionary that associates keys with values, as shown in Figure 6.6. The dictionary is implemented as a chained hash table that incrementally rehashes keys into a larger bucket array as it grows. Redis' dictionary structure is polymorphic (using the `type` field and a table of function pointers), allowing reuse of generic dictionary code in different contexts. For the main dictionary, the keys are length-prefixed strings, and the values are `redisObject` objects, that store the type and encoding of the data and a reference count.

We check several types of invariants for Redis:

- Type-safety of pointers, including `void*` fields whose type is determined by the dictionary type, or redisObject type. This invariant ensures that the polymorphic dictionary type is used correctly, and it is safe to

Corruption Type	Result			
	Caught	Safe	Latent	Crashed
Stray Write	429	70	1	0
Corrupt Write	463	37	0	0
Missed Write	350	150	0	0

Figure 6.7: Redis corruption results

dereference pointers, e.g., when checking other invariants.

- Uniqueness of pointers to the bucket array, dictionary entries, and key strings, which prevents memory leaks, dangling pointers, and cycles in linked lists.
- Reference counts on redisObjects match the number of pointers to that object.
- Hash table size field is less than or equal to the allocated size of the bucket array.
- Keys are hashed into the correct bucket.
- Key strings are immutable until they are destroyed.
- Rehashing state is consistent, preventing elements from being skipped during rehashing. This inconsistency may not be detected until rehashing completes, but the invariant ensures that data is not lost.

To support these invariants, we add an auxiliary field to a key string relating it to its containing dictionary entry, and to a dictEntry relating it to its containing dictionary. The invariants are implemented in 436 C lines of code, including 56 assertions.

Corruption Results

Figure 6.7 shows our corruption results for Redis protected by Ingot. No crashes were observed, showing that Ingot can catch consistency-violating corruptions before an operation becomes durable. A crash may occur before commit, in which case, the undo log allows us to revert any changes made by the buggy transaction, preserving the consistent, durable state that has previously been checked and committed.

The safe corruptions fell into several categories: missed writes to a timestamp field, a corruption that altered the "rehash index" of a dictionary in a safe way (causing the program to re-inspect some empty buckets in the hash table), and corruption of a string value (not a key). Adding a checksum would allow detecting stray writes to these values.

We saw one latent corruption caused by a stray write to a field in the dictionary structure that is used to track currently open iterators on the dictionary. This corruption would affect application correctness, but since Redis' iterators are not persistent, we do not consider it to be a persistent corruption.

Performance Results

We use the redis-benchmark tool to measure the performance of GET and SET operations. GET operations have minimal impact on database state and so they do not trigger invariant checks except when rehashing. However, they do incur some overhead associated with a transaction commit when Ingot determines whether there is anything to check. In our experiments, we separate these operations so that we can quantify the overheads of read and write operations.

The Redis server process is allocated two cores (one for the main thread, and one for the background Atlas log cleaning thread). The benchmark is run on a separate client machine, and we ensure that the throughput is

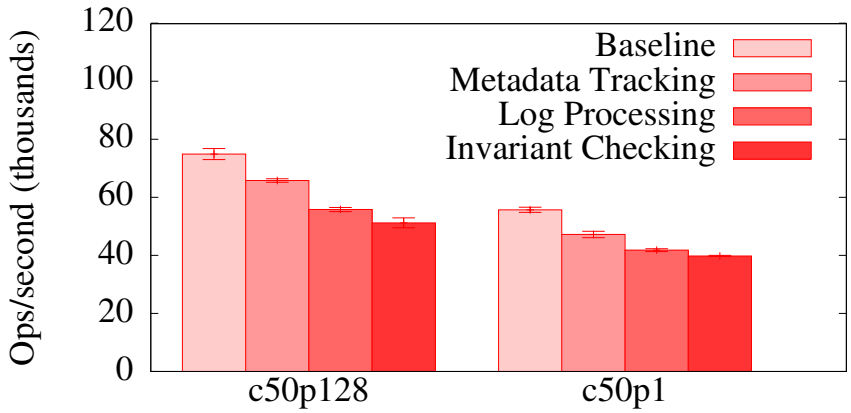


Figure 6.8: Redis SET throughput (50 connections, pipeline depth of 128 and 1)

not limited by the client or the network. We inject load on the server by running the client with 50 simultaneous connections, ensuring that there are always pending requests on the server. We use two configurations in the benchmark tool, a single command per request, and a batch of 128 commands per request ("pipelining", in Redis terminology). While the single command is a more typical workload, the second reduces per-command overhead by amortizing the impact of OS and network latency.

Figure 6.8 shows the throughput of SET operations in Redis for the two benchmark configurations (pipeline depth of 128 and 1). In each configuration, we show two main results: *Baseline*, which consists of persistence-enabled Redis using the ATLAS framework, and *Invariant Checking*, which uses the entire Ingot infrastructure to checks differential invariants in Redis. We also show two intermediate points, *Metadata Tracking*, and *Log Processing*, that break down the overheads of Ingot.

In the c50p128 configuration, Ingot-Redis has 68% of the baseline throughput (51200 ops/s versus 74900 ops/s, or roughly 46% overhead). In the c50p1 configuration, Ingot-Redis has 70% of the baseline throughput (39800 ops/s versus 55700 ops/s, or roughly 43% overhead). Inverting the throughput numbers for the c50p128 configuration yields an average time of 12us per operation for the baseline configuration, and 18us for Ingot. The 6us Ingot overhead can be broken down into three parts, metadata tracking, log processing, and invariant checking, each of which takes roughly 2us.

The first two overheads are related to applications written in an unmanaged language. Metadata tracking inserts and deletes records into Ingot's allocation metadata whenever objects are created or destroyed in persistent memory. Ingot must pay the cost of tracking heap allocations because the C runtime does not provide run-time type information. Log processing consists of two components, constructing a consistency point, which is inexpensive for single-threaded applications, and iterating through the log entries, interpreting the types of objects and their field. We need to interpret types because in the presence of bugs, we cannot statically determine the object or field type that was written through a (potentially corrupt!) pointer. Instead we use the Ingot allocation metadata to look up each address by its containing allocation.

We confirmed these results using the `perf` monitoring tool in Linux. In the c50p128 configuration, more than 70% of Ingot's 6us overhead results from "metadata" operations: creating or deleting allocation metadata, querying allocation metadata while parsing the log (i.e., to determine what object and field is at a given address), or querying allocation metadata while checking the invariants (e.g., to check that the type of a pointer is correct). More efficient methods for tracking run-time type information, and the mapping from an address to its containing allocation (e.g., using shadow memory, or integration with the memory allocator), will help reduce these costs.

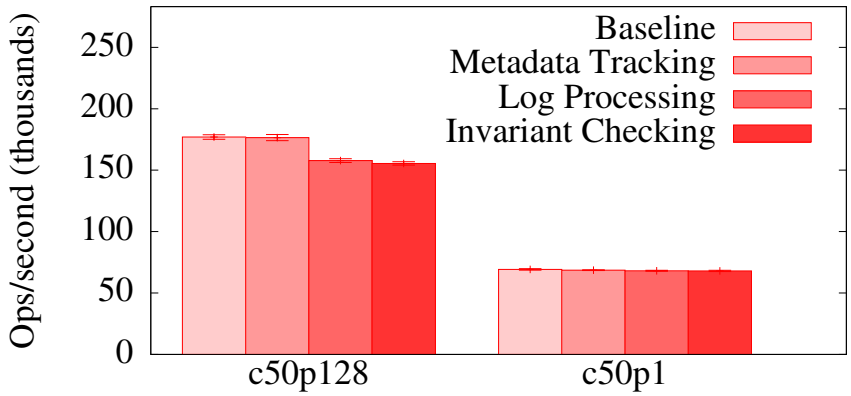


Figure 6.9: Redis GET throughput (50 connections, pipeline depth of 128 and 1)

	Throughput (ops/s)			
	c50p1		c50p128	
Keys	Global	Ingot	Global	Ingot
100	52219	43087	78468	61477
1000	19422	42758	76769	56646
10000	1963	40997	44412	54436
100000	123	39821	5463	51727

Figure 6.10: Checking a *single* global invariant versus checking all invariants with Ingot

The median latency for the baseline is 83ms, while median latency for Ingot is 120ms. This 37ms difference in latency can be accounted for by the 6us per operation overhead: 6us x 128 operations per request x 50 concurrent requests = 38.4ms. Ingot has a higher impact on tail latencies. At the 99th percentile, baseline latency is 137ms while Ingot latency is 204ms. This increase is due to variance in checking time (e.g., time spent in memory allocators, or the need to check more invariants when the main Redis dictionary is being rehashed).

Figure 6.9 shows the performance of GET operations. Since there are few metadata updates, metadata tracking is close to the baseline throughput. Similarly, there are few writes, so invariant checking has minimal overheads. The main overhead is due to log processing, which performs end-of-transaction processing, including acquiring a lock, determining whether there are other threads to roll back, and creating and cleaning up various book-keeping data structures.

Next, we illustrate the difference between checking a global consistency invariant and checking differential invariants in Redis. We measured the performance of checking a *single* global consistency invariant that is invoked after every SET operation in baseline persistent Redis. This invariant implements a check of the hashes of all the elements in the dictionary, by iterating through the dictionary in a loop and asserting that each entry is correct. Figure 6.10 shows Redis throughput with increasing number of keys in the hash table. As the data set size grows, performance tumbles. Ingot exceeds the performance of the global check with roughly 1000 keys in the dictionary. Additionally, Ingot is not just checking the hash table invariant differentially, but all the other invariants that we've implemented for Redis as well.

Given the excellent scalability of differential invariant checking as data sizes grow, the importance of protecting persistent data structures from corruption, and the opportunities to reduce the overheads in Ingot's metadata management, we believe our results show that Ingot is viable in production settings.

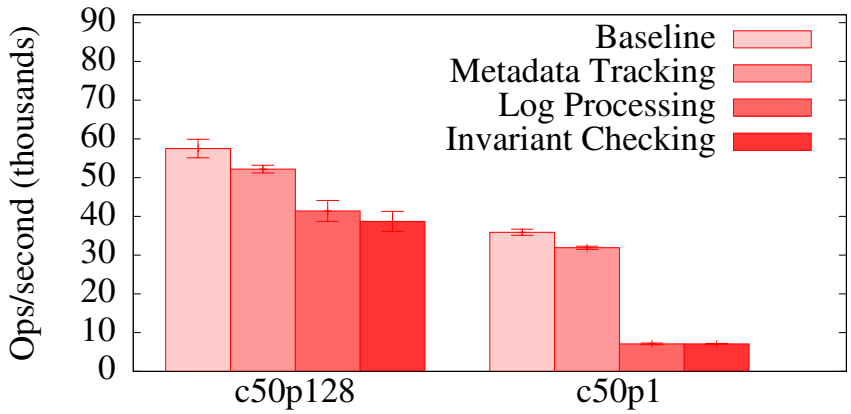


Figure 6.11: Thredis SET throughput (50 connections, pipeline depth of 128 and 1)

Thredis Results

Our multi-threaded version of Redis is based on Thredis, which dispatches requests to worker threads and uses fine-grained locking for synchronization.

Figure 6.11 shows the throughput of SET operations in Thredis for the two benchmark configurations. When compared with Redis (Figure 6.11), the most significant difference in Ingot's Thredis SET performance is the overhead introduced by the log processing step. This occurs because Ingot needs to pause worker threads in Thredis when a thread is committing. To construct a consistency point, Ingot temporarily rolls back the updates of the paused threads before performing the checks.

Most of this overhead occurs because Linux does not provide a low-overhead means to suspend a subset of the threads of an application. Since we use signal handlers to suspend and resume the other threads, generating a consistency point requires two invocations of pthread_kill per thread, per commit. Furthermore, once the signal is set, Ingot spins on a set of flag variables to determine that all the signals have been received. A single system call that suspended or resumed other threads and then returned would reduce this overhead greatly. An alternative would to be to provide an API so that the application could "cooperatively" indicate consistency points during which all threads would be quiescent.

We also performed corruption tests on Thredis, similar to the Redis tests. We found two latent corruptions. In both cases, an empty directory was corrupted, for which we had not written invariants. The tests caused one crash, due to a missed write to the rehashing index of the dictionary. This write was supposed to set it to a sentinel value (-1), indicating no rehashing should occur, which was not captured in our invariants. The missing invariant is that when the rehash destination table is null, then the index must be set to the sentinel value.

Chapter 7

Future Work

This thesis presents many opportunities for future work. The following sections detail some specific areas: atomicity & durability in persistent memory, invariant specification, operation-specific invariants, the applicability of run-time checking to testing and program analysis, and performance.

7.1 Location Invariants for Persistent Memory

Like crash consistency mechanisms for block storage, location invariants also apply to crash consistency mechanisms for persistent memory. Unlike a journaling file system using redo logging, ATLAS undo logging requires ordering between each log entry, because the target location is updated immediately after logging the previous value. Because a processor may reorder these writes, ATLAS issues a write barrier after each log request. Our checking does not require these barriers, but we depend on them for recovery to a known-good state after the crash.

To implement location invariant checking for ATLAS we would need to intercept not only the writes which the application makes to persistent memory, but also all writes to ATLAS' persistent log region. In order to achieve the same guarantees as Recon + Location Invariants operating in isolation from the operating system, we would need hardware which snoops on CPU writes to the persistent log and heap regions, ensuring that writes to each are correctly ordered. Hardware snooping could also tighten the fault model of Ingot, allowing us to guarantee consistency even in the face of self-modifying code and CPU bugs.

7.2 Higher-level Invariant Specification

Converting global consistency invariants into their differential versions that cover every possible combination of updates is difficult to do correctly. One avenue of future work is to automatically derive the differential invariants and generate much of the necessary code from a higher level specification. Many consistency invariants can be expressed as first order logic queries (e.g., in SQL [34]), and it should be possible to apply techniques like differential relational calculus mechanically to produce the differential invariants. We would like to develop a tool to automate this process, ensuring that the complete list of differential checks is generated from a logical specification.

First order queries are unable to fully express consistency properties of recursive data structures. A potential specification language for consistency invariants which go beyond first-order logic is Datalog, a declarative lang-

uage similar to Prolog, but restricted to fixed-point queries on finite data sets, which are guaranteed to terminate. We envision that Datalog queries could be used express application-specific global consistency properties. Future work would be to leverage existing techniques for incremental Datalog evaluation to build a compiler from Datalog queries to C which executes directly on the application's data structures.

7.3 Operation Invariants

The Recon and Ingot systems are geared towards ensuring consistency of data structures, by checking consistency, atomicity, and durability invariants. All of these types of invariants are independent of any particular operation being performed. This leaves open the question of whether we can use run time checking to check more operation-specific invariants. For example, an operation which creates a new directory should not result in the removal of a directory entry. By checking invariants beyond just consistency, it might be possible to make guarantees closer to that of a fully verified storage system without incurring the same engineering and performance costs.

Checking operation invariants differs from checking consistency because it requires the checking system to know which operations were supposed to have been performed on the data being checked. This presents a challenge in high concurrency systems. Future work could look into whether it is feasible to check operation invariants without completely serializing operations.

For in-memory applications, a possible solution this problem is to integrate runtime checking with a persistence framework that implements *transactional memory*, which hides the effects of concurrent operations and only allows them to commit if they do not interfere with each other.

7.4 Run-time Checking for Testing and Program Analysis

Some tools for static analysis and symbolic execution use assertions as a way to determine that a program is behaving incorrectly. Run-time invariant checking provides the framework to efficiently make assertions about consistency, and so it could potentially be used as a target for automated testing or model checking. The goal would be to search for an execution path or set of inputs which drives the system in to an inconsistent state, revealing bugs that allow the program to reach that state.

Compared to checking invariants globally, the use of differential invariants as a target for program analysis may make it much more efficient to check larger, more complex program states. Further work is needed to implement differential invariant checking in an environment suitable for symbolic execution or static analysis.

7.5 Performance

Users are often willing to sacrifice reliability for performance, at least until they lose data. When NetApp incorporated run-time integrity checking into their WAFL file system, they limited the properties they checked based on performance (i.e., so that the performance penalty of the new checking was obscured by other performance improvements in that release of their operating system).

The most straightforward way to improve perfomance in the Ingot and Recon systems is optimization of spots in the existing implementation that were written with simplicity more in mind than performance. For instance, our Recon implementation uses a set differencing operation frequently, especially for checking Btrfs invariants. Each set is fully constructed by traversing file system metadata before the difference is calculated. However, in most instances we expect that not only is the difference between sets small, we expect to see elements appear in a

similar order. By alternating the construction of the old and new sets with the computation of the difference, we can keep the sizes of the intermediate sets small, saving both memory and time. In Ingot, more than 30% of CPU time is spent updating and querying the allocation metadata. We chose a Judy array for simplicity, but alternate approaches may have better tradeoffs. For instance, shadow memory has constant query time; integration with the memory allocator may also yield performance improvements.

The interleaving of concurrent operations may result in there being no point in wall clock time when all objects are consistent, unless all executing threads come to a quiescent state. This thesis explores one solution to this problem, which is to perform checking on a serialized log, but this may incur too great a cost for applications like high-concurrency databases. Alternative approaches to explore include asynchronous checking on object or page-granularity snapshots, application-provided consistency points and barriers, and forwarding streams of updates to checking on another core in parallel.

Ingot and Recon both aim to take a "hands off" approach to the application programming model, and are designed to work in environments with unmanaged code (e.g., C or C++). These design decisions impact performance. For instance, in a managed language (e.g., Java), the data type and field being written to can be determined statically, because pointers cannot be corrupted by programmer error, and pointer arithmetic is forbidden. An Ingot-like system could record all of the pertinent type information at the time the write occurs, rather than processing the log at the end of the transaction to determine what changed.

Chapter 8

Conclusions

This thesis describes how we can use run-time invariant checking to protect data from corruption originating at high levels in the storage stack. In concert with methods like replication and checksumming for protection against corruption in the *lower* levels of the storage stack, we can make guarantees of data structure consistency in file systems and other storage applications.

Consistency is crucial for the correctness of subsequent operations, and our focus on consistency also makes our approach *practical*, for several reasons. It allows us to isolate the checking code from the application internals and other volatile state, because the checker only needs to access the persistent state and pending updates. This isolation makes it feasible to run the checker in a separate address space, which protects the checking process from corruption due to a malfunctioning application. Consistency is also independent of concurrent interleavings of different operations, and so updates from multiple operations can be grouped together and checked simultaneously. Finally, consistency is easier to specify than correctness, but encompasses all of the assumptions about the previous state that an operation is allowed to make. The atomicity and durability properties that we check using *location invariants* in Chapter 5 are also essential for maintaining consistency after a crash or power loss. These three aspects make it possible to apply run-time invariant checking to existing systems, without re-writing their already mature and performant code.

Our approach addresses the *when*, *what*, and *how* of run-time invariant checking. The fundamental requirement of our approach is that the application is *crash consistent*, which ensures that there is a suitable point in logical time *when* it is safe to check the invariants (a consistency point). For in-memory applications, it suffices that the application is data-race free and protects its persistent data structures with locks; existing tools (e.g., ATLAS) can be used to instrument the application for crash consistency. The consistency point not only provides us a point in time when we *can* check invariants, but also an opportunity to check the invariants before the (potentially corrupt) writes take effect.

Every file system, database, or other storage application has its own data structures, and hence its own application-specific consistency invariants. Because of the scale of data in persistent storage, we emphasize the necessity of checking incrementally. Our systems check updates for consistency using differential invariants, which have been derived from global invariants that apply to the entire persistent state. Differential invariants use a snapshot of the old and new states to determine precisely what needs to be checked in order to ensure that the global invariants are not violated by the update. They rely on the consistency of the previous state, and so our checking relies on storage systems using other mechanisms to ensure that the checked data is preserved.

Our evaluation shows that the overheads of incremental checking are reasonable. For block storage systems,

the performance overhead of runtime consistency checking is largely masked by device latency. For persistent memory, we see a more substantial overhead, from 10%-50% in our Redis benchmarks. These numbers are also significantly affected by the choice of persistent memory framework adopted, and this is an opportunity for future improvement as more efficient frameworks are developed.

Bibliography

[1] E. E. Aftandilian, S. Z. Guyer, M. Vechev, and E. Yahav. Asynchronous assertions. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 275–288, 2011.

[2] A. V. Aho and J. D. Ullman. Universality of data retrieval languages. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '79, pages 110–119, 1979.

[3] J. Arnold and M. F. Kaashoek. Ksplice: automatic rebootless kernel updates. In *Proc. of the ACM SIGOPS European Conference on Computer Systems (Eurosys)*, pages 187–198, 2009.

[4] L. N. Bairavasundaram, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, G. R. Goodson, and B. Schroeder. An analysis of data corruption in the storage stack. *Transactions of Storage*, 4(3):1–28, 2008.

[5] L. N. Bairavasundaram, S. Sundararaman, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Tolerating file-system mistakes with envyfs. In *Proc. of the USENIX Annual Technical Conference*, June 2009.

[6] S. Behrens. Btrfs: runtime integrity check tool, Nov. 2011. `http://lwn.net/Articles/466493`.

[7] J. Bonwick and B. Moore. ZFS - The Last Word in File Systems. `http://opensolaris.org/os/community/zfs/docs/zfs_last.pdf`.

[8] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe. *ACM Transactions on Information and System Security*, 12:1–38, 2008.

[9] J. C. M. Carreira, R. Rodrigues, G. Candea, and R. Majumdar. Scalable testing of file system checkers. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 239–252, New York, NY, USA, 2012. ACM.

[10] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. In *Proc. of the ACM OOPSLA*, pages 433–452, 2014.

[11] F. Chen and G. Roşu. Mop: an efficient and generic runtime verification framework. In *Proc. of the ACM OOPSLA*, pages 569–588, 2007.

[12] H. Chen, D. Ziegler, T. Chajed, A. Chlipala, M. F. Kaashoek, and N. Zeldovich. Using crash hoare logic for certifying the fscq file system. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 18–37, 2015.

[13] V. Chidambaram, T. Sharma, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Consistency without ordering. In *Proc. of the USENIX Conference on File and Storage Technologies (FAST)*, Feb. 2012.

[14] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. NV-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 105–118, 2011.

[15] E. F. Codd. Relational completeness of data base sublanguages. In *Database Systems*, Courant Computer Symposia Series 6, pages 65–98. Prentice-Hall, 1972.

[16] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. Better I/O through byte-addressable, persistent memory. In *Proc. of the Symposium on Operating Systems Principles (SOSP)*, pages 133–146, 2009.

[17] J. Corbett. Towards better testing. LWN.net, March 2014. https://lwn.net/Articles/591985/.

[18] H. Custer. *Inside the Windows NT File System*. Microsoft Press, 1994.

[19] A. Danial. CLOC – Count Lines of Code. `http://cloc.sourceforge.net/`.

[20] B. Demsky and M. C. Rinard. Goal-directed reasoning for specification-based data structure repair. *IEEE Transactions on Software Engineering*, 32(12):931–951, 2006.

[21] T. Do, T. Harter, Y. Liu, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. HARDFS: Hardening HDFS with selective and lightweight versioning. In *Proc. of the USENIX Conference on File and Storage Technologies (FAST)*, Feb. 2013.

[22] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro. No compromises: Distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*, pages 54–70, 2015.

[23] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System software for persistent memory. In *Proc. of the ACM SIGOPS European Conference on Computer Systems (Eurosys)*, pages 1–15, 2014.

[24] Filebench version 1.4.9, 2011. `http://filebench.sourceforge.net`.

[25] D. Fryer, M. Qin, J. Sun, K. W. Lee, A. D. Brown, and A. Goel. Checking the integrity of transactional mechanisms. *Trans. Storage*, 10(4):17:1–17:23, Oct. 2014. doi: `http://doi.acm.org/10.1145/2675113`.

[26] D. Fryer, K. Sun, R. Mahmood, T. Cheng, S. Benjamin, A. Goel, and A. D. Brown. Recon: Verifying file system consistency at runtime. *ACM Transactions on Storage*, 8(4):15:1–15:29, Dec. 2012. doi: `http://doi.acm.org/10.1145/2385603.2385608`.

[27] G. R. Ganger, M. K. McKusick, C. A. N. Soules, and Y. N. Patt. Soft updates: a solution to the metadata update problem in file systems. *ACM Transactions on Computer Systems*, 18(2):127–153, 2000.

[28] O. R. Gatla, M. Hameed, M. Zheng, V. Dubeyko, A. Manzanares, F. Blagojevic, C. Guyot, and R. Mateescu. Towards robust file system checkers. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies*, FAST'18, pages 105–121, Berkeley, CA, USA, 2018. USENIX Association.

[29] P. Godefroid, M. Y. Levin, and D. Molnar. Automated whitebox fuzz testing. In *Proc. of the Network and Distributed System Security Symposium*, 2008.

[30] D. Griffin. jbd: correctly unescape journal data blocks, Mar. 2008. `http://kerneltrap.org/mailarchive/git-commits-head/2008/3/20/1206404/thread`.

[31] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, SIGMOD '95, pages 328–339, New York, NY, USA, 1995. ACM.

[32] H. S. Gunawi, V. Prabhakaran, S. Krishnan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Improving file system reliability with I/O shepherding. In *Proc. of the Symposium on Operating Systems Principles (SOSP)*, pages 293–306, 2007.

[33] H. S. Gunawi, A. Rajimwale, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. SQCK: A declarative file system checker. In *Proc. of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Dec. 2008.

[34] H. S. Gunawi, C. Rubio-González, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and B. Liblit. EIO: Error Handling is Occasionally Correct. In *Proceedings of the Sixth USENIX Conference on File and Storage Technologies (FAST '08)*, San Jose, CA, February 2008.

[35] A. Gupta and I. S. Mumick. *Maintenance of Materialized Views: Problems, Techniques, and Applications*, pages 145–157. MIT Press, 1999.

[36] R. Hagmann. Reimplementing the Cedar file system using logging and group commit. In *Proc. of the Symposium on Operating Systems Principles (SOSP)*, Nov. 1987.

[37] V. Henson, A. van de Ven, A. Gud, and Z. Brown. Chunkfs: Using divide-and-conquer to improve file system reliability and repair. In *Proc. of the Workshop on Hot Topics in System Dependability (HotDep)*, 2006.

[38] D. Hitz, J. Lau, and M. Malcolm. File system design for an NFS file server appliance. In *Proc. of the USENIX Annual Technical Conference*, 1994.

[39] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel. InkTag: Secure applications on an untrusted operating system. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 265–278, New York, NY, USA, 2013. ACM.

[40] T. C.-H. Hsu, H. Brügner, I. Roy, K. Keeton, and P. Eugster. NVthreads: Practical persistence for multi-threaded applications. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 468–482, 2017.

[41] Intel Corporation. Pmdk. `http://pmem.io/pmdk`.

[42] A. Kalia, M. Kaminsky, and D. G. Andersen. FaSST: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram RPCs. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 185–201, 2016.

[43] E. Kang and D. Jackson. Formal modeling and analysis of a flash filesystem in alloy. In *Proceedings of the 1st International Conference on Abstract State Machines, B and Z*, ABZ '08, pages 294–308, 2008.

[44] J. Kara. ext4: Always journal quota file modifications, June 2010. http://www.kerneltrap.org/mailarchive/linux-ext4/2010/6/2/6884775.

[45] J. Kara. jbd: Write journal superblock with WRITE_FUA after checkpointing, Apr. 2012. https://git.kernel.org/cgit/linux/kernel/git/tytso/ext4.git/commit/?id=fd2cbd4dfa3db477dd6226d387d3f1911d36a6a9.

[46] K. Keeton. The Machine: An architecture for memory-centric computing. Workshop on Runtime and Operating Systems for Supercomputers (ROSS), June 2015. http://www.mcs.anl.gov/events/workshops/ross/2015/slides/ross2015-keeton.pdf.

[47] S. R. Kleiman. Vnodes: An architecture for multiple file system types in Sun UNIX. In *USENIX Summer*, volume 86, pages 238–247, 1986.

[48] G. Kudrjavets, N. Nagappan, and T. Ball. Assessing the relationship between software assertions and faults: An empirical investigation. In *Proceedings of the 17th International Symposium on Software Reliability Engineering*, ISSRE '06, pages 204–212, Washington, DC, USA, 2006. IEEE Computer Society.

[49] H. Kumar, Y. Patel, R. Kesavan, and S. Makam. High performance metadata integrity protection in the WAFL copy-on-write file system. In *Proc. of the USENIX Conference on File and Storage Technologies (FAST)*, Feb. 2017.

[50] P. Lantz, S. Dulloor, S. Kumar, R. Sankaran, and J. Jackson. Yat: A validation framework for persistent memory software. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 433–438, Philadelphia, PA, 2014. USENIX Association.

[51] M. Liu, M. Zhang, K. Chen, X. Qian, Y. Wu, W. Zheng, and J. Ren. DudeTM: Building durable transactions with decoupling for persistent memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 329–343, 2017.

[52] L. Lu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and S. Lu. A study of Linux file system evolution. In *Proc. of the USENIX Conference on File and Storage Technologies (FAST)*, Feb. 2013.

[53] A. Ma, C. Dragga, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. ffsck: The fast file system checker. In *Proc. of the USENIX Conference on File and Storage Technologies (FAST)*, Feb. 2013.

[54] P. Macko, M. Seltzer, and K. A. Smith. Tracking back references in a write-anywhere file system. In *Proc. of the USENIX Conference on File and Storage Technologies (FAST)*, 2010.

[55] C. Mason, Nov. 2011. https://git.kernel.org/cgit/linux/kernel/git/tytso/ext4.git/commit/?id=387125fc722a8ed432066b85a552917343bdafca.

[56] A. Memaripour, A. Badam, A. Phanishayee, Y. Zhou, R. Alagappan, K. Strauss, and S. Swanson. Atomic in-place updates for non-volatile main memories with Kamino-Tx. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 499–512, 2017.

[57] M. Mesnier, F. Chen, T. Luo, and J. B. Akers. Differentiated storage services. In *Proc. of the Symposium on Operating Systems Principles (SOSP)*, pages 57–70, 2011.

[58] D. T. Meyer and W. J. Bolosky. A study of practical deduplication. In *Proc. of the 9th USENIX Conference on File and Storage Technologies (FAST)*, pages 1–13, Feb. 2011.

[59] R. Miller. Joyent Services Back After 8 Day Outage, Jan. 2008. `http://www.datacenterknowledge.com/archives/2008/01/21/joyent-services-back-after-8-day-outage/`.

[60] J. Mohan, A. Martinez, S. Ponnapalli, P. Raju, and V. Chidambaram. Finding crash-consistency bugs with bounded black-box crash testing. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 33–50, Carlsbad, CA, 2018. USENIX Association.

[61] L. V. Orman. Differential relational calculus for integrity maintenance. *IEEE Trans. on Knowl. and Data Eng.*, 10(2):328–341, Mar. 1998.

[62] J. H. Perkins, S. Kim, S. Larsen, S. P. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. C. Rinard. Automatically patching errors in deployed software. In *Proc. of the Symposium on Operating Systems Principles (SOSP)*, pages 87–102, 2009.

[63] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Model-based failure analysis of journaling file systems. In *Proc. of the IEEE Dependable Systems and Networks (DSN)*, pages 802–811, 2005.

[64] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. IRON file systems. In *Proc. of the Symposium on Operating Systems Principles (SOSP)*, pages 206–220, 2005.

[65] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: Treating bugs as allergies — a safe method to survive software failures. In *Proc. of the Symposium on Operating Systems Principles (SOSP)*, pages 235–248, Oct. 2005.

[66] C. Reichenbach, N. Immerman, Y. Smaragdakis, E. E. Aftandilian, and S. Z. Guyer. What can the GC compute efficiently?: A language for heap assertions at GC time. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 256–269, 2010.

[67] Uber Technologies Inc. Why Uber engineering switched from postgres to mysql, July 2016.

[68] O. Rodeh, J. Bacik, and C. Mason. BTRFS: The Linux B-tree filesystem. *ACM Trans. Storage*, 9(3):9:1–9:32, Aug. 2013.

[69] Rubio-González, Cindy, Gunawi, H. S., B. Liblit, Arpaci-Dusseau, R. H., Arpaci-Dusseau, and A. C. Error propagation analysis for file systems. In *Proc. of the ACM SIGPLAN conference on programming language design and implementation (PLDI)*, pages 270–280, 2009.

[70] E. Sandeen. ext4: fix unjournaled inode bitmap modification, Oct. 2012. `https://lwn.net/Articles/521819/`.

[71] A. Shankar and R. Bodík. Ditto: Automatic incrementalization of data structure invariant checks (in Java). In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 310–319, 2007.

[72] H. Sigurbjarnarson, J. Bornholt, E. Torlak, and X. Wang. Push-button verification of file systems via crash refinement. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 1–16, Savannah, GA, 2016. USENIX Association.

[73] A. Silverstein. Application note: Simulating content addressable memory. `http://judy.sourceforge.net/examples/content_addressable_memory.pdf`, Sept. 2001.

[74] G. Sivathanu, S. Sundararaman, and E. Zadok. Type-safe disks. In *Proc. of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 15–28, 2006.

[75] M. Sivathanu, V. Prabhakaran, F. I. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Semantically-smart disk systems. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 73–88, 2003.

[76] O. Sokolsky, U. Sammapun, I. Lee, and J. Kim. Run-time checking of dynamic properties. *Electronic Notes in Theoretical Computer Science*, 144:91–108, May 2006.

[77] S. Sundararaman, S. Subramanian, A. Rajimwale, A. C. Arpaci-dusseau, R. H. Arpaci-dusseau, and M. M. Swift. Membrane: Operating system support for restartable file systems. In *Proc. of the USENIX Conference on File and Storage Technologies (FAST)*, 2010.

[78] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS file system. In *Proc. of the USENIX Annual Technical Conference*, pages 1–14, 1996.

[79] G. Trubetskoy. Thredis. `http://thredis.org/`, 2012.

[80] T. Ts'o. Re: Apparent serious progressive ext4 data corruption bug in 3.6.3, Oct. 2012. `https://lkml.org/lkml/2012/10/23/690`.

[81] S. C. Tweedie. Journalling the ext2fs filesystem. In *Proceedings of the 4th Annual Linux Expo*, May 1998.

[82] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. In *Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 91–104, 2011.

[83] J. Yang, C. Sar, and D. Engler. EXPLODE: a lightweight, general system for finding serious storage system errors. In *Proc. of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[84] J. Yang, C. Sar, P. Twohey, C. Cadar, and D. Engler. Automatically generating malicious disks using symbolic execution. In *Proc. of the IEEE Symposium on Security and Privacy*, pages 243–257, 2006.

[85] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using model checking to find serious file system errors. *ACM Transactions on Computer Systems*, 24(4):393–423, 2006.

[86] Y. Zhang, A. Rajimwale, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. End-to-end data integrity for file systems: a ZFS case study. In *Proc. of the USENIX Conference on File and Storage Technologies (FAST)*, 2010.